---

title: T-Swap Audit Report

author: Khant Wai Yan Aung (SnavOhBurmaa) date: 19, 4, 2026

T-Swap Audit Report

Prepared by: Khant Wai Yan Aung (SnavOhBurmaa) (https://www.linkedin.com/in/khant-wai-yan-ohburmaa/)

About Me

Hi, I'm Khant Wai Yan Aung (SnavOhBurmaa). I'm focusing on smart contract security audit. Currently studying at Rangsit University(CS).

Risk Classification

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

Audit Details

The findings described in this document correspond the following repository:

```
https://github.com/Cyfrin/5-t-swap-audit
```

Scope

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

Issues found

Severity	Count
High	5
Medium	3
Low	3
Informational	2
Gas	1
Total	**14**

Findings

High

[H-1] `_swap` gives free tokens every 10 swaps and breaks the `x * y = k` invariant

Location: `src/TSwapPool.sol:46, 47, 399-403`

Description

`_swap` keeps a counter `swap_count`. Every 10th swap, it sends an extra `1e18` of the output token to the caller for free. This bonus is outside the constant-product math, so the pool reserves silently lose value while the `x * y = k` invariant is never restored.

```
   uint256 private swap_count = 0;
   uint256 private constant SWAP_COUNT_MAX = 10;
   ...
   function _swap(IERC20 inputToken, uint256 inputAmount, IERC20
outputToken, uint256 outputAmount) private {
       ...
@>     swap_count++;
@>     if (swap_count >= SWAP_COUNT_MAX) {
@>         swap_count = 0;
@>         outputToken.safeTransfer(msg.sender,
1_000_000_000_000_000_000);
@>     }
       ...
   }
```

Risk

Likelihood: High. Any swapper triggers the bonus every 10 swaps with no special conditions. A trader can loop small swaps to farm the bonus on purpose.

Impact: High. Liquidity providers lose funds on every 10th swap; the protocol's core invariant is violated, which is the design guarantee LPs rely on.

Proof of Concept

Add this test to `test/unit/TSwapPool.t.sol`:

```
function testInvariantBrokenBy10thSwap() public {
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 100e18);
    pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
    vm.stopPrank();

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

    uint256 poolWethBefore = weth.balanceOf(address(pool));
    uint256 startWeth = weth.balanceOf(user);

    for (uint256 i = 0; i < 10; i++) {
        pool.swapExactInput(poolToken, 1e17, weth, 0,
uint64(block.timestamp));
    }

    uint256 endWeth = weth.balanceOf(user);
    uint256 poolWethAfter = weth.balanceOf(address(pool));
    uint256 totalInput = 10 * 1e17; // 10 swaps of 0.1 poolToken
each

    console.log("Total poolToken sent in: ", totalInput);
    console.log("WETH user received:      ", endWeth - startWeth);
    console.log("Pool WETH before:        ", poolWethBefore);
    console.log("Pool WETH after:         ", poolWethAfter);
    console.log("Pool WETH drained:       ", poolWethBefore -
poolWethAfter);

    // 1 poolToken in total should net ~0.9 WETH, but the 10th-
swap bonus
    // pushes the user above 1 WETH received.
    assertGt(endWeth - startWeth, 1e18);
}
```

Run:

```
forge test --match-test testInvariantBrokenBy10thSwap -vv
```

Expected console output:

```
Total poolToken sent in:   1000000000000000000   (1 poolToken)
WETH user received:        1987158034397321802   (~1.98 WETH)
Pool WETH before:          10000000000000000000
Pool WETH after:           9801284196560267 8198
Pool WETH drained:         1987158034397321802   (~1.98 WETH for 1
poolToken input — bonus applied)
```

Recommended Mitigation

Delete the bonus. If a reward program is really wanted, pay from a separate treasury, not pool reserves.

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

Also remove `swap_count` and `SWAP_COUNT_MAX` state variables.

[H-2] `getInputAmountBasedOnOutput` uses the wrong fee scalar, overcharging users by ~10x

Location: `src/TSwapPool.sol:293-295`

Description

T-Swap's fee is `0.3%`. `getOutputAmountBasedOnInput` correctly uses `997 / 1000`:

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

But `getInputAmountBasedOnOutput` uses `10000` instead of `1000`:

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

This pumps the computed input by roughly 10x.

Risk

Likelihood: High. Every call through `swapExactOutput` and `sellPoolTokens`, which wraps it triggers the bug.

Impact: High. Direct loss of user funds: traders pay ~10x the intended amount for exact-output swaps.

Proof of Concept

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

    // Correct fee math (997/1000) would require ~1.014e18
poolToken
    // to receive 1 WETH out. The buggy 10000/997 scalar bumps
this ~10x.
    uint256 expectedInputCorrect = 1014492753623188406; //
~1.014e18

    vm.startPrank(user);
    poolToken.mint(user, 100e18); // make sure user has enough to
be drained
    uint256 startPT = poolToken.balanceOf(user);
    poolToken.approve(address(pool), type(uint256).max);
    pool.swapExactOutput(poolToken, weth, 1e18,
uint64(block.timestamp));
    uint256 spentPT = startPT - poolToken.balanceOf(user);

    console.log("WETH requested out:      ", uint256(1e18));
    console.log("Expected poolToken cost: ",
expectedInputCorrect);
    console.log("Actual poolToken cost:   ", spentPT);
    console.log("Overpayment (wei):       ", spentPT -
expectedInputCorrect);

    assertGt(spentPT, 5e18); // way above the ~1.014e18 correct
cost
}
```

Run:

```
forge test --match-test testSwapExactOutputOvercharges -vv
```

Expected console output:

```
WETH requested out:         1000000000000000000   (1 WETH)
Expected poolToken cost:    1014492753623188406   (~1.014
```

```
poolToken)
Actual poolToken cost:    10152284263959390862   (~10.15
poolToken)
Overpayment (wei):         9137791510336202456   (~9.14 poolToken
— user paid ~10x)
```

Recommended Mitigation

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

[H-3] `sellPoolTokens` calls the wrong function and passes the wrong argument

Location: `src/TSwapPool.sol:365-375`

Description

`sellPoolTokens` should let the user sell a **known amount** of pool tokens and receive WETH. That is an exact-input swap. But the code calls `swapExactOutput` and passes `poolTokenAmount` as the `outputAmount`, so the user accidentally asks the pool for `poolTokenAmount` WETH, and the input cost is whatever the pool quotes made worse by H-2

```
   function sellPoolTokens(
       uint256 poolTokenAmount
   ) external returns (uint256 wethAmount) {
       return
@>         swapExactOutput(
               i_poolToken,
               i_wethToken,
@>             poolTokenAmount,   // used as outputAmount — wrong
               uint64(block.timestamp)
           );
   }
```

Risk

Likelihood: High. Any user calling `sellPoolTokens` hits this.

Impact: High. Combined with H-2 the caller pays far more than intended and receives a fixed WETH amount that has nothing to do with what they meant to sell. Direct loss of funds.

Proof of Concept

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

    vm.startPrank(user);
    poolToken.mint(user, 100e18);
    poolToken.approve(address(pool), type(uint256).max);
    uint256 startPT = poolToken.balanceOf(user);
    uint256 startWeth = weth.balanceOf(user);

    // User intent: "I want to sell 1 poolToken and receive some
WETH."
    pool.sellPoolTokens(1e18);

    uint256 gainedWeth = weth.balanceOf(user) - startWeth;
    uint256 spentPT = startPT - poolToken.balanceOf(user);

    console.log("poolToken user wanted to sell: ", uint256(1e18));
    console.log("poolToken actually spent:      ", spentPT);
    console.log("WETH received (forced output): ", gainedWeth);

    // Bug: the 1e18 was treated as outputAmount, so user got
exactly 1 WETH
    // back but paid an unrelated (and, due to H-2, inflated)
amount of poolToken.
    assertEq(gainedWeth, 1e18);
}
```

Run:

```
forge test --match-test testSellPoolTokensWrongAmount -vv
```

Expected console output:

```
poolToken user wanted to sell:   1000000000000000000   (1
poolToken)
poolToken actually spent:       10152284263959390862   (~10.15
poolToken)
WETH received (forced output):   1000000000000000000   (1 WETH —
forced, not a real "sell")
```

Recommended Mitigation

Call `swapExactInput` and add a `minWethToReceive` slippage parameter.

```
   function sellPoolTokens(
        uint256 poolTokenAmount,
```

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

[H-4] `swapExactInput` never assigns its named return value, always returns `0`

Location: `src/TSwapPool.sol:298-324`

Description

`swapExactInput` declares `returns (uint256 output)` but shadows it with a local `outputAmount` and never writes to `output`. Solidity returns the default zero value.

```
    function swapExactInput(
        IERC20 inputToken,
        uint256 inputAmount,
        IERC20 outputToken,
        uint256 minOutputAmount,
        uint64 deadline
    )
        public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
@>      returns (uint256 output)       // declared
    {
        ...
@>      uint256 outputAmount =
getOutputAmountBasedOnInput(...);  // local shadow
        ...
        _swap(inputToken, inputAmount, outputToken, outputAmount);
        // `output` is never assigned, so return value is 0
    }
```

Risk

Likelihood: High. Fires on every call.

Impact: Medium. Contracts, routers, and UIs that rely on the return value to decide next steps (refunds, multi-hop routing, bookkeeping) get a zero and can mis-handle the flow. Potential loss for integrators depending on how the value is used.

Proof of Concept

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

    vm.startPrank(user);
    uint256 startWeth = weth.balanceOf(user);
    poolToken.approve(address(pool), 10e18);

    uint256 returned = pool.swapExactInput(
        poolToken, 10e18, weth, 0, uint64(block.timestamp)
    );
    uint256 actualReceived = weth.balanceOf(user) - startWeth;

    console.log("Return value from swapExactInput: ", returned);
    console.log("Real WETH received by user:       ",
actualReceived);

    // The function returns 0 even though real WETH was
transferred.
    assertEq(returned, 0);
    assertGt(actualReceived, 0);
}
```

Run:

```
forge test --match-test testSwapExactInputReturnsZero -vv
```

Expected console output:

```
Return value from swapExactInput:   0
Real WETH received by user:         9066108938801491315   (~9.07
WETH — ignored by callers)
```

Recommended Mitigation

Assign the named return directly:

```
-   uint256 outputAmount = getOutputAmountBasedOnInput(
+   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);
```

[H-5] `LiquidityAdded` event emits arguments in the wrong order

Location: `src/TSwapPool.sol:52-56` (declaration), `src/TSwapPool.sol:196` (emission)

Description

The event is declared as:

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

But the emission passes `poolTokensToDeposit` in the `wethDeposited` slot and `wethToDeposit` in the `poolTokensDeposited` slot:

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

Risk

Likelihood: High. Fires on every non-initial deposit.

Impact: Medium. Off-chain indexers will attribute the wrong amounts to the wrong token. Silent data corruption that is hard to catch in review.

Proof of Concept

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

```
function testLiquidityAddedEventIsSwapped() public {
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 100e18);

    // Contract will emit the two amounts in the wrong slots.
    vm.expectEmit(true, false, false, true);
    emit LiquidityAdded(liquidityProvider, 50e18, 100e18); //
swapped on purpose
    pool.deposit(100e18, 1, 50e18, uint64(block.timestamp));
}
```

Run:

```
forge test --match-test testLiquidityAddedEventIsSwapped -vv
```

Recommended Mitigation

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

Medium

[M-1] `deposit` accepts a `deadline` parameter but never enforces it

Location: `src/TSwapPool.sol:113-122`

Description

`deposit` takes `uint64 deadline` but does not apply the `revertIfDeadlinePassed` modifier. A pending deposit transaction can be mined minutes or hours late when the pool ratio has moved against the LP, and the contract will still accept it.

```
   function deposit(
       uint256 wethToDeposit,
       uint256 minimumLiquidityTokensToMint,
       uint256 maximumPoolTokensToDeposit,
@>     uint64 deadline                 // never checked
   )
       external
       revertIfZero(wethToDeposit)
       returns (uint256 liquidityTokensToMint)
```

Risk

Likelihood: High. Any LP using a deadline expects it to be enforced; every stale tx path hits this.

Impact: Medium. LPs may be forced into deposits at unfavorable ratios when txs sit in the mempool; classic MEV/delay risk. The `maximumPoolTokensToDeposit` guard partially compensates but does not replace a deadline.

Proof of Concept

```
function testDepositIgnoresDeadline() public {
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 100e18);

    uint64 pastDeadline = uint64(block.timestamp - 1);

    console.log("block.timestamp: ", block.timestamp);
    console.log("deadline passed: ", pastDeadline);

    // Deposit should revert because the deadline has already
passed, but it succeeds.
    pool.deposit(100e18, 1, 100e18, pastDeadline);

    console.log("LP tokens minted: ",
pool.balanceOf(liquidityProvider));
    assertEq(pool.balanceOf(liquidityProvider), 100e18);
}
```

Run:

```
forge test --match-test testDepositIgnoresDeadline -vv
```

Expected console output:

```
block.timestamp:    1
deadline passed:    0                       (deadline <
block.timestamp)
LP tokens minted:   100000000000000000000    (deposit still went
through — guard ignored)
```

Recommended Mitigation

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

```
+       revertIfDeadlinePassed(deadline)
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
```

[M-2] `swapExactOutput` has no max-input (slippage) protection

Location: `src/TSwapPool.sol:337-358`

Description

`swapExactOutput` computes the required input based on current reserves and pulls whatever amount comes out of `getInputAmountBasedOnOutput`. There is no `maxInputAmount` argument, so a price move or a sandwich attack between simulation and execution can force the user to pay much more than they expected.

```
   function swapExactOutput(
       IERC20 inputToken,
       IERC20 outputToken,
       uint256 outputAmount,
       uint64 deadline
   )
       public
       revertIfZero(outputAmount)
       revertIfDeadlinePassed(deadline)
       returns (uint256 inputAmount)
   {
       ...
       inputAmount = getInputAmountBasedOnOutput(outputAmount,
inputReserves, outputReserves);
@>     // no cap — whatever the pool says, the user pays
       _swap(inputToken, inputAmount, outputToken, outputAmount);
   }
```

Risk

Likelihood: High. Any public mempool tx is sandwich-able.

Impact: Medium. Users can pay significantly more input than intended; no direct theft from the pool, but loss from the user's wallet.

Proof of Concept

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

```
    // Quote BEFORE price move — what the victim expects to pay.
    uint256 expectedPay = pool.getInputAmountBasedOnOutput(
        1e18,
        poolToken.balanceOf(address(pool)),
        weth.balanceOf(address(pool))
    );

    // Attacker front-runs to move the price (sandwich step 1).
    address attacker = makeAddr("attacker");
    poolToken.mint(attacker, 500e18);
    vm.startPrank(attacker);
    poolToken.approve(address(pool), type(uint256).max);
    pool.swapExactInput(poolToken, 500e18, weth, 0,
uint64(block.timestamp));
    vm.stopPrank();

    // Victim still asks for exactly 1 WETH out — no
maxInputAmount to cap cost.
    vm.startPrank(user);
    poolToken.mint(user, 100e18);
    poolToken.approve(address(pool), type(uint256).max);
    uint256 before = poolToken.balanceOf(user);
    pool.swapExactOutput(poolToken, weth, 1e18,
uint64(block.timestamp));
    uint256 paid = before - poolToken.balanceOf(user);

    console.log("Expected poolToken cost (pre-attack):  ",
expectedPay);
    console.log("Actual poolToken cost (post-attack):   ", paid);
    console.log("Extra paid due to missing slippage:    ", paid -
expectedPay);

    assertGt(paid, expectedPay * 2);
}
```

Run:

```
forge test --match-test testSwapExactOutputNoSlippageGuard -vv
```

Expected console output:

```
Expected poolToken cost (pre-attack):   10152284263959390862
(~10.15 — inflated by H-2 scalar)
Actual poolToken cost (post-attack):    60974025974025974026
(~60.97 — far above expected)
Extra paid due to missing slippage:     50821741710066583164
(~50.82 poolToken lost to sandwich)
```

Recommended Mitigation

Add a `maxInputAmount` parameter:

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

[M-3] `PoolFactory::createPool` uses `.name()` instead of `.symbol()`

Location: `src/PoolFactory.sol:52`

Description

The LP token symbol is built from the underlying token's `name()` instead of its `symbol()`:

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

For a pool of a token with name `"USD Coin"` and symbol `"USDC"`, the LP symbol becomes `"tsUSD Coin"` (with a space) instead of the intended `"tsUSDC"`.

Risk

Likelihood: High. Every pool created hits this.

Impact: Medium. Wallets, explorers, and UIs render the LP token with a broken symbol. ERC-20 symbols are expected to be short, space-free identifiers.

Proof of Concept

```
function testFactoryUsesNameForSymbol() public {
    ERC20Mock token = new ERC20Mock(); // name: "ERC20Mock",
symbol: "E20"
    address poolAddr = factory.createPool(address(token));
    TSwapPool p = TSwapPool(poolAddr);
```

```
    console.log("Underlying token name:   ", token.name());
    console.log("Underlying token symbol: ", token.symbol());
    console.log("LP token name:           ", p.name());
    console.log("LP token symbol (buggy): ", p.symbol());

    // LP symbol should be "tsE20" (from symbol). It is
"tsERC20Mock" (from name).
    assertEq(p.symbol(), "tsERC20Mock");
}
```

Run:

```
forge test --match-test testFactoryUsesNameForSymbol -vv
```

Expected console output:

```
Underlying token name:     ERC20Mock
Underlying token symbol:   E20
LP token name:             T-Swap ERC20Mock
LP token symbol (buggy):   tsERC20Mock      (expected "tsE20")
```

Recommended Mitigation

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

Low

[L-1] `PoolFactory::createPool` has no zero-address check on `tokenAddress`

Location: `src/PoolFactory.sol:47-58`

Description

`createPool` does not validate that `tokenAddress != address(0)`:

```
   function createPool(address tokenAddress) external returns
(address) {
       if (s_pools[tokenAddress] != address(0)) {
           revert PoolFactory__PoolAlreadyExists(tokenAddress);
       }
       ...
       TSwapPool tPool = new TSwapPool(tokenAddress, i_wethToken,
liquidityTokenName, liquidityTokenSymbol);
```

```
        ...
    }
```

A `createPool(address(0))` call would deploy a pool whose pool-token side is the zero address. The call fails later when `.name()` is invoked on zero, but the failure mode is noisy.

Risk

Likelihood: Low. Requires caller to pass zero.

Impact: Low. Waste of gas and a confusing revert; no theft.

Recommended Mitigation

```
+   if (tokenAddress == address(0)) revert
        PoolFactory__ZeroAddress();
```

[L-2] `PoolFactory` constructor has no zero-address check on `wethToken`

Location: `src/PoolFactory.sol:40-42`

Description

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

Deploying with `wethToken == address(0)` bricks every future pool: the WETH side cannot receive or send tokens.

Risk

Likelihood: Low. Operator error at deploy time.

Impact: Medium if it happens. Every deployed pool is broken.

Recommended Mitigation

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

[L-3] `PoolFactory::createPool` allows `tokenAddress == i_wethToken`

Location: `src/PoolFactory.sol:47`

Description

Nothing prevents `createPool(i_wethToken)`. A WETH/WETH pool has no economic purpose and will behave strangely when `_isUnknown` checks are applied inside `_swap`.

Risk

Likelihood: Low. Requires caller to pass WETH as the pool token.

Impact: Low. Wasted deployment, user confusion, potential DoS surface on `_swap`.

Recommended Mitigation

```
+   if (tokenAddress == i_wethToken) revert
        PoolFactory__SameTokens();
```

Info

[I-1] `swapExactInput` visibility should be `external` instead of `public`

Location: `src/TSwapPool.sol:298-305`

`swapExactInput` is not called internally. Marking it `external` saves gas on calldata handling.

```
-   function swapExactInput(...) public
+   function swapExactInput(...) external
```

[I-2] `PoolCreated` event has no indexed fields

Location: `src/PoolFactory.sol:35`

```
event PoolCreated(address tokenAddress, address poolAddress);
```

Neither field is `indexed`, so off-chain tooling cannot efficiently filter `PoolCreated` by token or pool.

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

Gas

[G-1] `_isUnknown` can be simplified to a single return

Location: `src/TSwapPool.sol:416-421`

```
   function _isUnknown(IERC20 token) private view returns (bool) {
       if (token != i_wethToken && token != i_poolToken) {
           return true;
       }
       return false;
   }
```

The `if/else` wrapping of a boolean expression is redundant. Inline the comparison:

```
   function _isUnknown(IERC20 token) private view returns (bool) {
-      if (token != i_wethToken && token != i_poolToken) {
-          return true;
-      }
-      return false;
+      return token != i_wethToken && token != i_poolToken;
   }
```

Small bytecode and gas win, plus clearer intent.