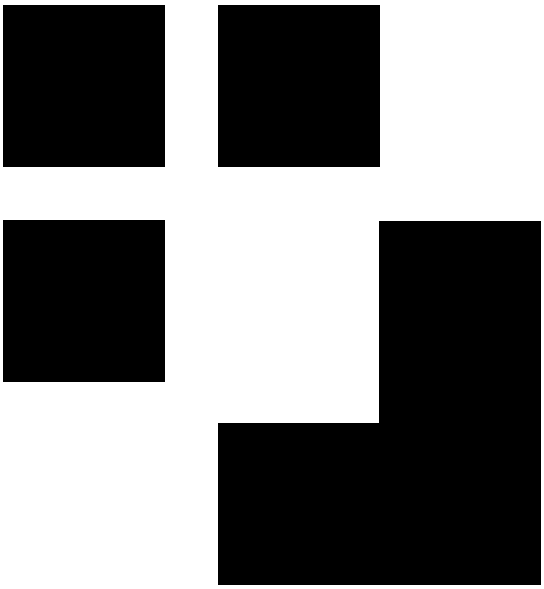

Protocol Audit Report

Version 1.0

Hexific.com

September 16, 2025

Protocol Audit Report

Hexific.com

September 16, 2025

Prepared by: Hexific Lead Security Researcher: - Febri Nirwana

Table of Contents

Protocol Summary

The TSwapPool is an automated market maker (AMM) smart contract designed to facilitate swaps between WETH and a designated pool token. It allows users to add or remove liquidity, swap tokens, and earn fees from trades. The protocol uses a constant product formula (x * y = k) to determine pricing and aims to provide a simple, gas-efficient interface for decentralized token swaps. TSwapPool is intended to be deployed alongside a PoolFactory contract, which manages the creation of new pools for different tokens.

Disclaimer

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

Risk Classification

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

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

Audit Details

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

```
f426f57731208727addc20adb72cb7f5bf29dc03
```

Scope

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

Roles

- Liquidity Providers: Users who deposit WETH and pool tokens to provide liquidity and earn fees from swaps.

- Traders/Swappers: Users who swap between WETH and pool tokens, paying fees to liquidity providers.
- Pool Deployers: Anyone can create new pools through the PoolFactory by calling `createPool()` with any ERC20 token address.

Executive Summary

The Hexific team conducted a security audit of the TSwap protocol smart contracts to evaluate their design, implementation, and resistance to common attack vectors. Our review identified several critical issues that could lead to significant financial losses for users and protocol insolvency.

The most severe findings include incorrect fee calculations that charge users 90% instead of 0.3%, missing slippage protection that exposes users to MEV attacks, and a protocol invariant-breaking mechanism that drains pool reserves. Additionally, we found function logic errors, missing deadline protections, and various code quality issues.

The protocol implements an automated market maker (AMM) design but contains fundamental flaws that make it unsafe for production use in its current state. Immediate remediation of the High severity findings is strongly recommended before any deployment.

Issues found

Severity	Number of Issues Found
High	4
Medium	1
Low	2
Gas	1
Info	8
Total	16

Findings

High

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

Description: This function is intended to calculate how much input tokens are needed to receive a specific amount of output tokens, applying a 0.3% fee (following the README). However, the fee calculation is implemented incorrectly. The function uses `* 10000` and `* 997` which creates super huge 90.3% fee.

Impact: - Users lose so much money by paying excessive amounts for their desired output - Protocol loses fee revenue due to incorrect fee calculation mechanism
- Breaks core AMM functionality and makes the protocol unusable for exact output swaps

Proof of Concept:

1. Correct fee calculation (from `getOutputAmountBasedOnInput`):

```
// 0.3% fee = 997/1000 (takes 99.7% of input)
uint256 inputAmountMinusFee = inputAmount * 997;
uint256 numerator = inputAmountMinusFee * outputReserves;
uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee;
```

2. Incorrect fee calculation (from `getInputAmountBasedOnOutput`):

```
// This calculates 90.03% fee instead of 0.3%!
// 997/10000 = 0.0997, meaning 1 - 0.0997 = 0.9003 = 90.03% fee
return ((inputReserves * outputAmount) * 10000) /
       ((outputReserves - outputAmount) * 997);
```

3. Mathematical proof:

    - Pool state: 100 WETH, 100 Pool Tokens
    - User wants: 10 WETH output

    With correct calculation: ~10.3 Pool Tokens needed With current broken calculation: ~101 Pool Tokens needed (10x more!)

Recommended Mitigation: Fix the fee calculation to match the correct formula used in `getOutputAmountBasedOnInput`:

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

```
3       uint256 inputReserves,
4       uint256 outputReserves
5   )
6       public
7       pure
8       revertIfZero(outputAmount)
9       revertIfZero(outputReserves)
10      returns (uint256 inputAmount)
11  {
12  -   return ((inputReserves * outputAmount) * 10000) /
13  -          ((outputReserves - outputAmount) * 997);
14  +   return ((inputReserves * outputAmount) * 1000) /
15  +          ((outputReserves - outputAmount) * 997);
16  }
```

[H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` causes users to potentially receive way fewer tokens

Description: Unlike `TSwapPool::swapExactInput()` which has a `minOutputAmount` parameter to protect users from unfavorable price movements, `TSwapPool::swapExactOutput()` has no `maxInputAmount` parameter. This means users could end up paying significantly more input tokens than they intended if the pool state changes between transaction submission and execution.

Impact: - Users can lose substantial funds by paying far more input tokens than expected due to price movements - No protection against MEV attacks where malicious actors front-run transactions to manipulate prices

Proof of Concept:

Attack scenario: - Pool state: 100 WETH, 100 Pool Tokens (1:1 ratio) - User wants exactly 10 WETH, expecting to pay ~10.3 Pool Tokens - Attacker front-runs with large trade, changing ratio to 100 WETH, 1000 Pool Tokens - User's transaction executes, now paying ~111 Pool Tokens instead of ~10.3 - User loses ~100 Pool Tokens due to lack of slippage protection

Recommended Mitigation: Add a `maxInputAmount` parameter for slippage protection:

```
1   function swapExactOutput(
2       IERC20 inputToken,
3       IERC20 outputToken,
4       uint256 outputAmount,
5   +   uint256 maxInputAmount,
6       uint64 deadline
7   )
8   ...
9   {
10      ...
11  +   if (inputAmount > maxInputAmount) {
```

```
12 +         revert TSwapPool__InputTooHigh(inputAmount, maxInputAmount);
13 +     }
14
15       _swap(inputToken, inputAmount, outputToken, outputAmount);
16   }
```

Also add the corresponding error:

```
1 + error TSwapPool__InputTooHigh(uint256 actual, uint256 max);
```

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

Description: The function is designed to allow users to sell a specific amount of pool tokens in exchange for WETH. However, it incorrectly uses `swapExactOutput()` instead of `swapExactInput()`. The function passes `poolTokenAmount` as the `outputAmount` parameter to `swapExactOutput()`, but `poolTokenAmount` represents the input amount (what the user wants to sell), not the output amount (what they want to receive).

Impact: Users will receive a completely wrong amount of WETH, potentially orders of magnitude different from expected

Proof of Concept:

1. What actually happens:
    - User calls `sellPoolTokens(100)` wanting to sell 100 pool tokens
    - Function calls `swapExactOutput` with `outputAmount = 100`
    - This means "give me exactly 100 WETH" instead of "I want to sell 100 pool tokens"
    - `swapExactOutput` calculates how many pool tokens needed to get 100 WETH
    - User ends up paying far more pool tokens than intended
2. Example scenario:
    - Pool state: 1000 WETH, 1000 Pool Tokens
    - User wants to sell 100 Pool Tokens (should get ~90 WETH with fees)
    - Current implementation: Tries to get exactly 100 WETH output
    - Calculates ~111 Pool Tokens needed as input
    - User loses 111 Pool Tokens instead of 100, gets 100 WETH instead of ~90

Recommended Mitigation: Use `swapExactInput` instead of `swapExactOutput`:

```
1 function sellPoolTokens(
2     uint256 poolTokenAmount
```

```
) external returns (uint256 wethAmount) {
    return
-       swapExactOutput(
+       swapExactInput(
            i_poolToken,
+           poolTokenAmount,
            i_wethToken,
-           poolTokenAmount,
+           0, // or implement a reasonable minimum
            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 `_swap` function includes a mechanism that gives users extra output tokens every 10 swaps (`SWAP_COUNT_MAX = 10`). When `swap_count` reaches this threshold, the function transfers an additional `1e17` tokens (0.1 ETH worth) to the user without taking any corresponding input tokens. This breaks the core AMM invariant of `x * y = k` because tokens are leaving the pool without maintaining the mathematical relationship that ensures proper pricing.

Impact: - The `x * y = k` invariant is fundamental to AMM pricing - breaking it causes incorrect exchange rates - Attackers can specifically target every 10th swap or do a lot of swaps with low amounts to extract maximum value

Proof of Concept:

The test shows that on the 10th swap, the user receives more tokens than they should: - Expected: User should only get `outputWeth * 10` (10e17) tokens - Actual: User gets `outputWeth * 10 + 1e17` due to the extra token mechanism

The invariant `x * y = k` is broken because: 1. Normal swap: Proper exchange maintaining `(x - output)* (y + input)= k` 2. 10th swap: `(x - output - EXTRA)* (y + input)!= k` where EXTRA = 1e17

PoC

```
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;
```

```
9
10      vm.startPrank(user);
11      poolToken.approve(address(pool), type(uint256).max);
12      poolToken.mint(user, 100e18); // give enough tokens to user
13      pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp)); // 1
14      pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp)); // 2
15      pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp)); // 3
16      pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp)); // 4
17      pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp)); // 5
18      pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp)); // 6
19      pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp)); // 7
20      pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp)); // 8
21      pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp)); // 9
22
23      int256 startingX = int256(weth.balanceOf(address(pool)));
24
25      pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp)); // 10
26      vm.stopPrank();
27
28      uint256 endingX = weth.balanceOf(address(pool));
29
30      int256 expectedDeltaX = -int256(outputWeth); // Should only lose
            outputWeth amount
31      int256 actualDeltaX = int256(endingX) - int256(startingX);
32      assertEq(actualDeltaX, expectedDeltaX);
33  }
```

```
1  Output:
2  [FAIL: assertion failed: -1100000000000000000 != -1000000000000000000]
       testInvariantBroken() (gas: 484530)
```

Recommended Mitigation: Remove the extra token incentive mechanism as it's fundamentally incompatible with AMM mathematics:

```
1  -   swap_count++;
2  -   if (swap_count >= SWAP_COUNT_MAX) {
3  -       swap_count = 0;
4  -       outputToken.safeTransfer(msg.sender, 1e17);
5  -   }
```

Also remove the related state variables:

```
- uint256 private swap_count = 0;
- uint256 private constant SWAP_COUNT_MAX = 10;
```

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:** Refers to the compiler `forge build`:

```
Warning (5667): Unused function parameter. Remove or comment out the
    variable name to silence this warning.
   --> src/TSwapPool.sol:123:9:
    |
123 |         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: The event is defined with parameters in the order (`address indexed liquidityProvider, uint256 wethDeposited, uint256 poolTokensDeposited`

) on lines 52-56. However, when the event is emitted in the `TSwapPool::_addLiquidityMintAndTransfer()` function on line 229, the parameters are passed in the wrong order: `LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit)`. This means the `wethDeposited` and `poolTokensDeposited` values are swapped.

Impact: Off-chain monitoring tools, indexers, and frontend applications that listen to this event will receive incorrect data

Proof of Concept: 1. Event definition (lines 52-56):

```
event LiquidityAdded(
    address indexed liquidityProvider,
    uint256 wethDeposited,        // Should be second parameter
    uint256 poolTokensDeposited   // Should be third parameter
);
```

2. Event emission (line 229):

```
emit LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit);
//                              ^^^^^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^
//                              This is pool tokens  This is WETH
//                              (should be 3rd)      (should be 2nd)
```

Recommended Mitigation: Fix the parameter order in the event emission:

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

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

Description: The function is declared with a return value `returns (uint256 output)` but never actually returns any value. In Solidity, when a function declares a return value but doesn't explicitly return anything, it returns the default value for that type (0 for uint256). This means callers will always receive 0 as the output amount, regardless of how much they actually received from the swap.

Impact: - **Low Impact**: Misleading data for external contracts or frontends and users trying to track their swap results programmatically

Proof of Concept: Refers to the compiler `forge build`:

```
Warning (5667): Unused function parameter. Remove or comment out the
    variable name to silence this warning.
    --> src/TSwapPool.sol:358:18:
     |
358 |         returns (uint256 output)
```

```
5         |
```

Recommended Mitigation: Return the calculated output amount:

```
1 {
2 +   return outputAmount;
3 }
```

Gas

[G-1] Public functions should be external when not called internally

Description: Several functions in `TSwapPool` are declared as **`public`** but are never called internally by the contract. Functions that are only called externally should be declared as `external` for better gas efficiency and clearer intent.

Impact: Slightly higher gas costs for external calls to public functions

Recommended Mitigation: Change public functions to external:

```
1 - function getOutputAmountBasedOnInput(...) public pure returns (
     uint256) {
2 + function getOutputAmountBasedOnInput(...) external pure returns (
     uint256) {
3
4 - function getInputAmountBasedOnOutput(...) public pure returns (
     uint256) {
5 + function getInputAmountBasedOnOutput(...) external pure returns (
     uint256) {
6
7 - function sellPoolTokens(...) public returns (uint256) {
8 + function sellPoolTokens(...) external returns (uint256) {
```

Information

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

Description: This error on line 24 is never used anywhere in the codebase. Dead code like unused errors increases the contract's bytecode size unnecessarily and can confuse developers about the contract's intended functionality.

Impact: - Increases contract deployment gas costs due to larger bytecode - Code maintainability - dead code can confuse developers and auditors - Potential for future bugs if developers assume this

error is functional

Recommended Mitigation: Remove the unused error definition:

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

[I-2] `PoolFactory::constructor` Lacking zero address check

Description: The constructor accepts a `wethToken` parameter and assigns it to the immutable variable `i_wethToken` without validating that the address is not `address(0)`. If `address(0)` is passed during deployment, the factory will be deployed with an invalid WETH token address, potentially causing all pool creation and functionality to fail.

Impact: - If deployed with `address(0)`, the entire factory becomes unusable - All attempts to create pools would likely fail when trying to interact with the zero address - Requires complete redeployment of the factory contract to fix

Recommended Mitigation: Add a zero address check in the constructor:

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

Also add the corresponding error definition:

```
+ error PoolFactory__InvalidAddress();
```

[I-3] `PoolFactory::createPool` should use `.symbol()` instead of `.name()` for token symbol

Description: In the `createPool()` function, when creating the liquidity token symbol on line 49, the code incorrectly uses `IERC20(tokenAddress).name()` instead of `IERC20(tokenAddress).symbol()`. Token symbols are meant to be short identifiers (typically 3-4 characters), while token names are longer descriptive names. Using the full token name for the symbol can result in excessively long and inappropriate symbols for the liquidity tokens.

Impact: - Liquidity tokens will have inappropriately long symbols that don't follow ERC20 conventions - Poor user experience - token symbols may be too long to display properly in wallets and interfaces - Inconsistency with standard DeFi practices where LP token symbols are short

Recommended Mitigation: Change line 49 to use `.symbol()` instead of `.name()`:

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

[I-4] `TSwapPool::constructor` lacks zero address check for token parameters

Description: The `TSwapPool` constructor accepts `poolToken` and `wethToken` addresses but doesn't validate that these addresses are not `address(0)`. If either parameter is accidentally set to the zero address during deployment, the pool will be deployed with invalid token addresses, making it completely non-functional.

Impact: - Pool becomes completely unusable if deployed with zero addresses - All swap and liquidity operations would fail when trying to interact with `address(0)` - Requires complete redeployment of the pool contract to fix - Potential loss of gas costs from failed deployment

Proof of Concept:

```
// Current constructor has no validation
constructor(
    address poolToken,
    address wethToken,
    string memory liquidityTokenName,
    string memory liquidityTokenSymbol
) ERC20(liquidityTokenName, liquidityTokenSymbol) {
    i_wethToken = IERC20(wethToken);      // No zero check
    i_poolToken = IERC20(poolToken);      // No zero check
}
```

Recommended Mitigation: Add zero address checks in the constructor:

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

[I-5] `TSwapPool::poolTokenReserves` variable unused and should be removed

Description: The function `poolTokenReserves()` is declared in the contract but is never called or used anywhere in the codebase. This represents dead code that unnecessarily increases the contract's bytecode size and gas costs.

Impact: - Increases contract deployment gas costs due to larger bytecode - Code bloat that can confuse developers and auditors - Maintenance overhead for unused functionality - **Proof of Concept:** The function exists on line 295 but has no references:

```
function poolTokenReserves() external view returns (uint256) {
    return i_poolToken.balanceOf(address(this));
}
```

Searching the codebase shows this function is never called internally or externally.

Recommended Mitigation: Remove the unused function:

```
- function poolTokenReserves() external view returns (uint256) {
-     return i_poolToken.balanceOf(address(this));
- }
```

[I-6] `TSwapPool::deposit` function violates CEI (Checks-Effects-Interactions) pattern

Description: The `deposit` function performs external token transfers before updating internal state. The CEI pattern recommends doing all checks first, then effects (state changes), and finally interactions (external calls). This pattern helps prevent reentrancy attacks and makes the code more secure and predictable.

Impact: Deviates from security best practices

Recommended Mitigation:

1. Reorganize the function to follow CEI pattern:

```
function deposit(...) external returns (uint256 liquidityTokensToMint)
    {
    ...
    else {
+       liquidityTokensToMint = wethToDeposit;
        _addLiquidityMintAndTransfer(
            wethToDeposit,
            maximumPoolTokensToDeposit,
            wethToDeposit
        );
-       liquidityTokensToMint = wethToDeposit;
```

```
11      }
12  }
```

1. Alternatively, return directly from the function after performing the external call:

```
1  function deposit(...) external returns (uint256) {
2      ...
3      else {
4          _addLiquidityMintAndTransfer(
5              wethToDeposit,
6              maximumPoolTokensToDeposit,
7              wethToDeposit
8          );
9  -       liquidityTokensToMint = wethToDeposit;
10 +       return wethToDeposit;
11     }
12 }
```

[I-7] `TSwapPool` magic numbers should be replaced with named constants in fee calculation functions

Description: The functions `getOutputAmountBasedOnInput` and `getInputAmountBasedOnOutput` use magic numbers 997 and 1000 without explanation. These numbers represent the fee calculation (0.3% fee = 997/1000), but using raw numbers makes the code less readable and maintainable.

Impact: - Code is harder to understand and maintain - Risk of introducing bugs when modifying fee calculations

Recommended Mitigation: Define named constants for fee calculations:

```
1  + uint256 private constant FEE_DENOMINATOR = 1000;
2  + uint256 private constant FEE_NUMERATOR = 997;
3
4  function getOutputAmountBasedOnInput(...) {
5  -   uint256 inputAmountMinusFee = inputAmount * 997;
6  +   uint256 inputAmountMinusFee = inputAmount * FEE_NUMERATOR;
7  -   uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee;
8  +   uint256 denominator = (inputReserves * FEE_DENOMINATOR) +
       inputAmountMinusFee;
9  }
10
11 function getInputAmountBasedOnOutput(...) {
12 -   return ((inputReserves * outputAmount) * 1000) /
13 -          ((outputReserves - outputAmount) * 997);
14 +   return ((inputReserves * outputAmount) * FEE_DENOMINATOR) /
15 +          ((outputReserves - outputAmount) * FEE_NUMERATOR);
16 }
```

[I-8] `TSwapPool::swapExactOutput` missing parameter documentation in natspec

Description: The `swapExactOutput` function is missing complete natspec documentation for its parameters. While the function has a basic description, the `@param` tags are missing, making it harder for developers to understand the function's parameters and their purposes.

Impact: - Poor developer experience when integrating with the contract - Harder to understand function parameters without reading the implementation

Recommended Mitigation: Add complete natspec documentation:

```
+ /// @param deadline The deadline for the transaction to be completed
    by
function swapExactOutput(
    IERC20 inputToken,
    IERC20 outputToken,
    uint256 outputAmount,
    uint64 deadline
) public revertIfDeadlinePassed(deadline) returns (uint256 inputAmount)
    {
```