Security Analysis Report

T-Swap Protocol Audit

Prepared by: Nayashi

Date: November 2024

[H-01]

[H-1] Core-Invariant Broken, _swap() function sends additional tokens to caller for every 10 swaps which causes imbalance of tokens in the pool

Description: The `_swap` function in TSwapPool.sol contains a critical vulnerability where it sends additional tokens to users every 10 swaps through a reward mechanism. This breaks the core constant product formula (x * y = k) that AMMs rely on for price stability.

Specifically, in the `_swap` function:

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

The code sends 1e18 (1 token) of the output token to the caller every 10 swaps. This additional token transfer:

1. Is not accounted for in the constant product formula calculations
2. Happens after the main swap but uses the same output token
3. Creates an artificial drain on the pool's reserves

Impact: - This vulnerability fundamentally breaks the AMM mechanism:

1. Mathematical Invariant Violation:
    - The constant product formula (x * y = k) is violated with every 10th swap

- Pool reserves become increasingly imbalanced over time

2. Economic Impact:

   - Protocol loses 1e18 tokens every 10 swaps without compensation
   - Creates an arbitrage opportunity where users can profit by timing their trades
   - Pool becomes increasingly depleted of output tokens

3. Systemic Risk:

   - As the pool becomes imbalanced, all price calculations become inaccurate
   - Affects all users, not just those receiving the bonus

Proof of Concept:

```
     function invariant_constantProductFormulaStaysSameX() public {
      assertEq(handler.actualDelta_X(), handler.expectedDelta_X());
}
    function invariant_constantProductFormulaStaysSameY() public {
    assertEq(handler.actualDelta_Y(), handler.expectedDelta_Y());
}
```

These invariant tests fail because the actual delta (change in reserves) does not match the expected delta calculated from the constant product formula. This occurs because the bonus token mechanism in `_swap()` sends out additional tokens that aren't accounted for in the formula calculations.

Recommended Mitigation:

1. Remove the bonus token mechanism entirely:

```
function _swap(
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 outputAmount
) private {
    if (_isUnknown(inputToken) || _isUnknown(outputToken) || inputToken ==
outputToken) {
        revert TSwapPool__InvalidToken();
    }

    emit Swap(msg.sender, inputToken, inputAmount, outputToken, outputAmount);

    inputToken.safeTransferFrom(msg.sender, address(this), inputAmount);
```

```
        outputToken.safeTransfer(msg.sender, outputAmount);
    }
```

2. If a reward mechanism is desired:
    - Implement it as a separate mechanism outside the core swap logic
    - Use a dedicated reward token instead of pool tokens
    - Ensure any rewards don't affect the core AMM math

[H-2] Deadline not implemented in `deposit()` function

Description: The `deposit()` function has a `deadline` parameter but fails to implement the `revertIfDeadlinePassed` modifier, unlike other core functions like `withdraw()` and `swap()`. This means that transactions can be executed after their intended deadline, potentially leading to unfavorable trades.

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

Impact: HIGH - This vulnerability can lead to:

1. Stale transactions being executed at unfavorable prices
2. Front-running opportunities where attackers can manipulate the pool before a stale deposit executes
3. Users unable to rely on deadline protection for their deposit transactions

Proof of Concept: The test `testDepositPassesDeadline` in TSwapPool.t.sol demonstrates this vulnerability

```
function testDepositPassesDeadline() public {
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 100e18);
    uint64 deadline = uint64(block.timestamp);

    // Advance time past deadline
    vm.warp(deadline + 1);
    vm.roll(block.number + 1);

    // This deposit should revert but succeeds
    pool.deposit(100e18, 100e18, 100e18, deadline);

    // Verify deposit succeeded despite deadline passing
```

```
    assertEq(pool.balanceOf(liquidityProvider), 100e18);
    assertEq(weth.balanceOf(address(pool)), 100e18);
    assertEq(poolToken.balanceOf(address(pool)), 100e18);
    assertGt(block.timestamp, deadline); // Proves deadline was passed
}
```

Recommended Mitigation: Add the `revertIfDeadlinePassed` modifier to the deposit function:

```
function deposit(
    uint256 wethToDeposit,
    uint256 minimumLiquidityTokensToMint,
    uint256 maximumPoolTokensToDeposit,
    uint64 deadline
)
    external
    revertIfZero(wethToDeposit)
    revertIfDeadlinePassed(deadline)  // Add this modifier
    returns (uint256 liquidityTokensToMint)
```

[H-3] Incorrect Fee Calculation in getInputAmountBasedOnOutput() Leads to Excessive Fees and hence causing the user to take dusts

Description: The `getInputAmountBasedOnOutput()` function incorrectly implements the fee calculation, resulting in users being charged a 91.3% fee instead of the intended 0.3% fee. This occurs because the fee multipliers (10000 and 997) are applied in reverse order compared to the standard implementation.

```
function getInputAmountBasedOnOutput(
    uint256 outputAmount,
    uint256 inputReserves,
    uint256 outputReserves
) public pure returns (uint256 inputAmount) {
    return
        ((inputReserves * outputAmount) * 10000) /    // @audit: fee multipliers are
reversed
        ((outputReserves - outputAmount) * 997);      // should be * 997 / * 1000
}
```

The current implementation results in:

- Fee = 10000/997 = ~10.03
- Effective fee rate = (10.03 - 1) * 100 = 90.3% fee

Correct implementation should be:

- Fee = 1000/997 = ~1.003

- Effective fee rate = (1.003 - 1) * 100 = 0.3% fee

Impact: HIGH - This vulnerability causes:

1. Users are charged 91.3% fees on their trades instead of 0.3%
2. Makes the protocol unusable as trades are economically unfeasible
3. Direct loss of user funds through excessive fees

Proof of Concept:

```
function testExcessiveFees() public {
    // Setup pool with 100 WETH and 100 tokens
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 100e18);
    pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
    vm.stopPrank();

    // Try to swap 1 WETH worth of tokens
    uint256 inputAmount = pool.getInputAmountBasedOnOutput(
        1e18,  // Want 1 token
        100e18,  // WETH reserve
        100e18   // Token reserve
    );

    // With 0.3% fee, should be roughly 1.003e18
    // But actually returns ~10.03e18
    assertGt(inputAmount, 10e18); // Will pass, showing massive fee
}
```

Recommended Mitigation: Correct the fee calculation by adjusting the multipliers:

```
function getInputAmountBasedOnOutput(
    uint256 outputAmount,
    uint256 inputReserves,
    uint256 outputReserves
) public pure returns (uint256 inputAmount) {
    return
        ((inputReserves * outputAmount) * 1000) /
        ((outputReserves - outputAmount) * 997);
}
```

This change will result in the intended 0.3% fee structure that is standard in most AMM implementations.

[H-4] Missing Slippage Protection in swapExactOutput() Allows Sandwich Attacks

Description: The `swapExactOutput()` function lacks a maximum input amount parameter for slippage protection. Unlike `swapExactInput()` which has `minOutputAmount`, this function does not have an equivalent `maxInputAmount` parameter to protect users from paying more than expected for their trades.

```
function swapExactOutput(
    IERC20 inputToken,
    IERC20 outputToken,
    uint256 outputAmount,
    uint64 deadline
) public returns (uint256 inputAmount) {
    // ... reserves calculation ...
    inputAmount = getInputAmountBasedOnOutput(
        outputAmount,
        inputReserves,
        outputReserves
    );

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

Impact: HIGH - This vulnerability allows:

1. Sandwich attacks where attackers can manipulate the price before the transaction
2. Users paying significantly more input tokens than intended
3. No upper bound on the price a user might pay

Proof of Concept:

```
 function testExcessiveFees() public {
        // Setup pool with 100 WETH and 100 tokens
        vm.startPrank(liquidityProvider);
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

    // Try to swap 1 WETH worth of tokens
    uint256 inputAmount = pool.getInputAmountBasedOnOutput(
        1e18,  // Want 1 token
        100e18,  // WETH reserve
        100e18   // Token reserve
    );


        assertGt(inputAmount, 10e18); // Will pass, showing massive fee
    }
    function testSandwichAttack() public {
```

```
    // Setup initial liquidity
    vm.startPrank(liquidityProvider);
    weth.approve(address(pool), 100e18);
    poolToken.approve(address(pool), 100e18);
    pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
    vm.stopPrank();

    // Attacker manipulates price upward
    weth.mint(attacker, 50e18);
    vm.startPrank(attacker);
    weth.approve(address(pool), 50e18);
    pool.swapExactInput(weth, 50e18, poolToken, 0, uint64(block.timestamp));
    vm.stopPrank();

    // Victim's transaction - wants 1 token
    weth.mint(user, 25e18);
    vm.startPrank(user);
    weth.approve(address(pool),25e18);

    uint256 inputAmount = pool.swapExactOutput(
        weth,
        poolToken,
        1e18,
        uint64(block.timestamp)
    );
    // User pays much more than expected due to price manipulation
        assertGt(inputAmount, 2e18); // Will pay more than 2 WETH for 1 token
        vm.stopPrank();
    }
```

Recommended Mitigation: Add a `maxInputAmount` parameter and check:

```
function swapExactOutput(
    IERC20 inputToken,
    IERC20 outputToken,
    uint256 outputAmount,
    uint256 maxInputAmount,  // Add this parameter
    uint64 deadline
) public returns (uint256 inputAmount) {
    uint256 inputReserves = inputToken.balanceOf(address(this));
    uint256 outputReserves = outputToken.balanceOf(address(this));

    inputAmount = getInputAmountBasedOnOutput(
        outputAmount,
        inputReserves,
        outputReserves
    );

    if (inputAmount > maxInputAmount) {
```

```
            revert TSwapPool__ExcessiveInputAmount(inputAmount, maxInputAmount);
        }

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

Based on the remaining audit tags and following the same report structure, here are the additional findings:

[L-1] LiquidityAdded Event Emits Parameters in Wrong Order

Description: The `_addLiquidityMintAndTransfer` function emits the `LiquidityAdded` event with parameters in the wrong order, causing incorrect information to be logged.

```
function _addLiquidityMintAndTransfer(
    uint256 wethToDeposit,
    uint256 poolTokensToDeposit,
    uint256 liquidityTokensToMint
) private {
    _mint(msg.sender, liquidityTokensToMint);
    emit LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit); // @audit
wrong order
}
```

Impact: LOW

1. Event logs show incorrect information
2. Off-chain systems may misinterpret deposit amounts
3. No direct financial impact, but affects protocol transparency

Proof of Concept:

```
function testLiquidityAddedEventOrder() public {
    vm.startPrank(user);
    weth.approve(address(pool), 1e18);
    poolToken.approve(address(pool), 2e18);

    vm.expectEmit(true, true, true, true);
    emit LiquidityAdded(user, 1e18, 2e18); // Expected order
    pool.deposit(1e18, 0, 2e18, uint64(block.timestamp)); // Actual emits in wrong
order
}
```

Recommended Mitigation:

```
function _addLiquidityMintAndTransfer(
    uint256 wethToDeposit,
    uint256 poolTokensToDeposit,
    uint256 liquidityTokensToMint
) private {
    _mint(msg.sender, liquidityTokensToMint);
    emit LiquidityAdded(msg.sender, wethToDeposit, poolTokensToDeposit); // Correct
order
    // ... rest of function
}
```

[I-1] Magic Numbers Should Be Named Constants

Description: The contract uses magic numbers (997, 1000, 10000) for fee calculations. These should be named constants for better readability and maintenance.

```
function getInputAmountBasedOnOutput() {
    return ((inputReserves * outputAmount) * 10000) /
        ((outputReserves - outputAmount) * 997);
}

function getOutputAmountBasedOnInput() {
    uint256 inputAmountMinusFee = inputAmount * 997;
    uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee;
}
```

Impact: INFORMATIONAL

- Reduces code readability
- Makes maintenance more difficult
- Increases chance of errors in future updates

Recommended Mitigation:

```
// At contract level
uint256 private constant FEE_NUMERATOR = 997;
uint256 private constant FEE_DENOMINATOR = 1000;
uint256 private constant PRECISION_FACTOR = 10000;

function getInputAmountBasedOnOutput() {
    return ((inputReserves * outputAmount) * PRECISION_FACTOR) /
        ((outputReserves - outputAmount) * FEE_NUMERATOR);
}
```

[I-2] Incomplete NatSpec Documentation

Description: Several functions are missing complete NatSpec documentation, including `swapExactInput()` and the `deadline` parameter in `swapExactOutput()`.

Impact: INFORMATIONAL

- Reduces code maintainability
- Makes integration more difficult
- May lead to incorrect usage

Recommended Mitigation: Add complete NatSpec for all functions:

```
/// @notice Swaps an exact amount of input tokens for a minimum amount of output
tokens
/// @param inputToken The token to swap from
/// @param inputAmount The exact amount of tokens to swap
/// @param outputToken The token to swap to
/// @param minOutputAmount The minimum amount of tokens to receive
/// @param deadline The timestamp after which the trade will revert
/// @return output The actual amount of output tokens received
function swapExactInput(
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 minOutputAmount,
    uint64 deadline
) external returns (uint256 output)
```

[I-3] Public Function Should Be External

Description: The `totalLiquidityTokenSupply()` function is marked as `public` but is never called internally. It should be marked as `external` for gas optimization.

```
function totalLiquidityTokenSupply() public view returns (uint256)
```

Impact: INFORMATIONAL

- Minor gas inefficiency
- No functional impact

Recommended Mitigation:

```
function totalLiquidityTokenSupply() external view returns (uint256)
```

[L-1] Incorrect Liquidity Token Symbol Construction in Pool Creation

Description: In the `createPool` function, the liquidity token symbol is incorrectly constructed using the token's name instead of its symbol:

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

This leads to potentially long and incorrect token symbols, as it uses the full name instead of the standard symbol format.

Impact: LOW

1. Incorrect token symbol display in wallets and interfaces
2. Possible truncation of long symbols
3. Deviates from standard token symbol conventions
4. No direct financial impact, but affects usability and standards compliance

Proof of Concept:

```
function testIncorrectTokenSymbol() public {
    // Setup a mock token with
    // name = "Mock Token"
    // symbol = "MT"
    address mockToken = address(new MockToken());

    // Create pool
    address pool = factory.createPool(mockToken);

    // Current result: "tsMock Token" (incorrect)
    // Expected result: "tsMT"
    string memory symbol = TSwapPool(pool).symbol();
    assertEq(symbol, "tsMock Token"); // This passes, showing the issue
}
```

Recommended Mitigation: Change the symbol construction to use the token's symbol instead of name:

```
function createPool(address tokenAddress) external returns (address) {
    if (s_pools[tokenAddress] != address(0)) {
        revert PoolFactory__PoolAlreadyExists(tokenAddress);
    }

    string memory liquidityTokenName = string.concat(
        "T-Swap ",
        IERC20(tokenAddress).name()
```

```
        );
        string memory liquidityTokenSymbol = string.concat(
            "ts",
            IERC20(tokenAddress).symbol()  // Use symbol() instead of name()
        );

        TSwapPool tPool = new TSwapPool(
            tokenAddress,
            i_wethToken,
            liquidityTokenName,
            liquidityTokenSymbol
        );

        s_pools[tokenAddress] = address(tPool);
        s_tokens[address(tPool)] = tokenAddress;
        emit PoolCreated(tokenAddress, address(tPool));
        return address(tPool);
    }
```

[I-1] Unused Error Definition in PoolFactory Contract

Description: The `PoolFactory` contract defines an error that is never used in the codebase:

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

This error is defined but not utilized in any function, creating dead code.

Impact: INFORMATIONAL

- Unnecessary code bloat
- Potential confusion for developers
- Slightly higher deployment gas cost
- No functional or security impact

Proof of Concept:

1. The error is defined at the top of the contract:

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

2. Searching through the entire codebase shows no usage:

```
// No instances of:
revert PoolFactory__PoolDoesNotExist();
```

Recommended Mitigation: Either:

1. Remove the unused error:

```
contract PoolFactory {
    error PoolFactory__PoolAlreadyExists(address tokenAddress);
    // Remove: error PoolFactory__PoolDoesNotExist(address tokenAddress);
    ...
}
```

2. Or implement it where appropriate, such as in the `getPool` function:

```
function getPool(address tokenAddress) external view returns (address) {
    address pool = s_pools[tokenAddress];
    if (pool == address(0)) {
        revert PoolFactory__PoolDoesNotExist(tokenAddress);
    }
    return pool;
}
```