TSwap Audit Report

vicenteaguilar.com

June, 2025

TSwap Audit Report

Version 1.0

vicenteaguilar.com

June 30, 2025

TSwap Audit Report

vicenteaguilar.com

June, 2025

Prepared by: Vicent00 Lead Auditors: - xxxxxxx

Table of Contents

Protocol Summary

TSwap is a decentralized exchange (DEX) protocol implementing an automated market maker (AMM) model with constant product formula (x * y = k). The protocol allows users to swap tokens and provide liquidity through LP token minting and burning mechanisms. The core contract TSwapPool.sol handles all swap operations, liquidity management, and fee collection.

Disclaimer

The Vicent00 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

Scope

The audit covered the TSwapPool.sol contract and its interactions with ERC20 tokens for swap operations and liquidity management.

Roles

- Lead Auditor: Vicent00
- Security Analyst: [To be filled]

Executive Summary

Issues found

Severity	Count
Critical	2
High	6
Medium	8
Low	8
Informational	11

Total Issues: 35

The audit revealed multiple critical vulnerabilities including a backdoor in the `_swap` function that allows complete pool drainage, reentrancy vulnerabilities, and various security issues that pose severe risks to the protocol's security and economic stability.

Findings

Critical

[C-01] Backdoor in _swap Function Allows Complete Pool Drainage

Description

The `_swap` function in `TSwapPool.sol` contains a hidden backdoor that rewards users with 1_000_000_000_000_000_000 tokens (1e18) every 10 swaps. This mechanism can be exploited infinitely to drain the pool.

Impact

- Complete pool drainage
- Economic loss for all liquidity providers
- Protocol collapse
- Severe reputation damage

Proof of Concept

```
function exploitBackdoor() public {
    // Perform 10 swaps to trigger the backdoor
    for (uint256 i = 0; i < 10; i++) {
        pool.swapExactInput(
            address(token),
            1, // Minimal swap amount
            address(weth),
            0, // No minimum output
            block.timestamp
        );
    }
    // User now has 1_000_000_000_000_000_000 extra tokens from the backdoor
}
```

Recommended mitigation

Immediate Action Required: 1. **Remove the backdoor mechanism entirely** - This is not a legitimate feature and serves no economic purpose 2. **Implement proper access controls** - Add role-based permissions for any future reward mechanisms 3. **Emergency pause functionality** - Implement ability to pause all operations immediately

Code Fix:

```
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);
}
```

Long-term Recommendations: - Implement formal verification for all mathematical operations - Add comprehensive monitoring and alerting systems - Establish bug bounty program with significant rewards - Conduct regular security audits by multiple firms

[C-02] Reentrancy Vulnerability in _addLiquidityMintAndTransfer

Description

The `_addLiquidityMintAndTransfer` function is vulnerable to reentrancy attacks due to state changes before external calls, allowing attackers to drain the pool.

Impact

- Complete pool drainage and fund theft
- Protocol collapse
- Severe reputation damage

Proof of Concept

```
// State change before external calls
_mint(msg.sender, liquidityTokensToMint);  // State change first
i_wethToken.safeTransferFrom(msg.sender, address(this), wethToDeposit);  // External call af
```

Recommended mitigation

Critical Fix - Follow Checks-Effects-Interactions Pattern:

```
function _addLiquidityMintAndTransfer(
    uint256 wethToDeposit,
    uint256 poolTokensToDeposit,
    uint256 liquidityTokensToMint
) private {
    // 1. INTERACTIONS - External calls first
    i_wethToken.safeTransferFrom(msg.sender, address(this), wethToDeposit);
    i_poolToken.safeTransferFrom(msg.sender, address(this), poolTokensToDeposit);

    // 2. EFFECTS - State changes last
    _mint(msg.sender, liquidityTokensToMint);
    emit LiquidityAdded(msg.sender, wethToDeposit, poolTokensToDeposit);
}
```

Additional Security Measures: - Add reentrancy guard modifier to all functions with external calls - Implement proper event logging for all state changes - Add comprehensive input validation - Consider using OpenZeppelin's ReentrancyGuard

High

[H-01] Arithmetic Overflow/Underflow in Backdoor Exploitation

Description

When the backdoor is exploited repeatedly, it causes arithmetic overflow/underflow in various calculations, leading to contract malfunction.

Impact

- Contract becomes unusable
- Loss of user funds
- Potential for additional vulnerabilities

Proof of Concept

Invariant tests show overflow errors:

```
[FAIL. Reason: arithmetic overflow/underflow] test_invariant_noOverflow()
```

Recommended mitigation

Primary Fix - Remove Backdoor: The root cause is the backdoor mechanism. Remove it entirely as described in C-01.

Secondary Fixes - Add Overflow Protection:

```
// Ensure Solidity 0.8+ is used for built-in overflow protection
// Add validation for division by zero
require(denominator > 0, "Division by zero");
require(numerator / denominator <= outputReserves, "Insufficient output reserves");
```

Testing Recommendations: - Implement comprehensive fuzzing tests - Add invariant testing for all mathematical operations - Test with extreme values and edge cases - Use formal verification tools

[H-02] Missing Token Address Validation in createPool

Description

The `createPool` function accepts any `tokenAddress` without validating if it's a legitimate ERC20 token, allowing malicious contracts to be integrated into the protocol.

Impact

- Malicious token integration
- Protocol compromise
- Reputation damage
- Financial loss

Recommended mitigation

Add Comprehensive Token Validation:

```
function createPool(address tokenAddress) external returns (address) {
    require(tokenAddress != address(0), "Invalid token address: zero address");
    require(tokenAddress != i_wethToken, "Cannot create pool with WETH");
    require(s_pools[tokenAddress] == address(0), "Pool already exists");

    // Validate ERC20 interface
    try IERC20(tokenAddress).totalSupply() returns (uint256 totalSupply) {
        require(totalSupply > 0, "Token has zero total supply");
    } catch {
        revert("Invalid ERC20 token");
    }

    // Create pool with validated token
    string memory liquidityTokenName = string.concat("T-Swap ", IERC20(tokenAddress).name()
    string memory liquidityTokenSymbol = string.concat("ts", IERC20(tokenAddress).symbol())

    TSwapPool tPool = new TSwapPool(tokenAddress, i_wethToken, liquidityTokenName, liquidity

    s_pools[tokenAddress] = address(tPool);
    s_tokens[address(tPool)] = tokenAddress;

    emit PoolCreated(tokenAddress, address(tPool));
    return address(tPool);
}
```

Additional Security Measures: - Implement token whitelist mechanism -

Add governance controls for pool creation - Implement delay mechanism for new pool creation - Add comprehensive logging and monitoring

[H-03] Missing Fee Constants and Incorrect Fee Calculation

Description

The contract uses hardcoded fee values instead of constants and has incorrect fee calculations in the `getInputAmountBasedOnOutput` function, leading to pricing errors.

Impact

- Incorrect pricing calculations
- Arbitrage opportunities
- User losses
- Poor maintainability

Recommended mitigation

Define Fee Constants and Fix Calculations:

```
// Add fee constants
uint256 private constant FEE_DENOMINATOR = 1000;
uint256 private constant FEE_NUMERATOR = 997; // 0.3% fee

// Fix getInputAmountBasedOnOutput function - use FEE_DENOMINATOR instead of 10000
return ((inputReserves * outputAmount) * FEE_DENOMINATOR) /
    ((outputReserves - outputAmount) * FEE_NUMERATOR);
```

Additional Improvements: - Add fee configuration mechanism for future flexibility - Implement fee validation to ensure reasonable bounds - Add comprehensive testing for fee calculations

[H-04] Division by Zero in Withdraw Function

Description

The `withdraw` function can revert due to division by zero when `totalLiquidityTokenSupply()` is 0, blocking all withdrawal operations.

Impact

- Function blocking
- Fund lock
- Protocol unusability

Recommended mitigation

Add Validation for Zero Liquidity Token Supply:

```
function withdraw(...) external {
    // Add check for zero liquidity token supply
    require(totalLiquidityTokenSupply() > 0, "No liquidity tokens exist");

    // ... rest of function
}
```

Additional Safety Measures: - Add emergency withdrawal mechanism for edge cases - Implement gradual withdrawal limits - Add comprehensive event logging

[H-05] Slippage Attack Vulnerability in swapExactOutput

Description

The `swapExactOutput` function lacks a maximum input amount parameter, making it vulnerable to slippage attacks where users can be forced to pay significantly more than expected.

Impact

- Excessive token consumption
- MEV vulnerability
- Poor user experience
- Economic loss

Recommended mitigation

Add Maximum Input Amount Parameter:

```
function swapExactOutput(
    IERC20 inputToken,
    IERC20 outputToken,
    uint256 outputAmount,
    uint256 maxInputAmount,  // Add maximum input protection
    uint64 deadline
) public returns (uint256 inputAmount) {
    // Calculate required input amount
    inputAmount = getInputAmountBasedOnOutput(outputAmount, inputReserves, outputReserves);

    // Add slippage protection
    require(inputAmount <= maxInputAmount, "Input amount exceeds maximum");

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

Additional Protection Measures: - Implement price oracle integration for additional validation - Add time-weighted average price (TWAP) checks - Add comprehensive slippage monitoring

Medium

[M-01] Invariant Violation in Constant Product Formula

Description

The constant product formula `x * y = k` is violated when the backdoor is triggered, leading to incorrect swap calculations and potential arbitrage opportunities.

Impact

- Incorrect token pricing
- Potential arbitrage attacks
- Loss of protocol integrity

Proof of Concept

```
[FAIL. Reason: assertion failed] test_invariant_constantProduct()
```

Recommended mitigation

Primary Fix - Remove Backdoor: Remove the backdoor mechanism as described in C-01 to prevent invariant violations.

Secondary Fix - Add Invariant Validation:

```
// Add invariant checking to all state-changing functions
function _validateInvariant(uint256 wethReserves, uint256 poolTokenReserves) private pure {
    require(wethReserves > 0, "WETH reserves must be positive");
    require(poolTokenReserves > 0, "Pool token reserves must be positive");
}
```

Additional Measures: - Implement comprehensive invariant testing - Add real-time monitoring of invariant violations - Implement automatic circuit breakers for invariant violations

[M-02] Liquidity Consistency Violation

Description

The relationship between LP token supply and actual pool liquidity is violated when the backdoor creates tokens out of thin air.

Impact

- LP token devaluation
- Unfair distribution of fees
- Loss of trust in the protocol

Proof of Concept

```
[FAIL. Reason: assertion failed] test_invariant_liquidityConsistency()
```

Recommended mitigation

Primary Fix - Remove Backdoor: Remove the backdoor mechanism as described in C-01 to prevent liquidity consistency violations.

Secondary Fix - Add Liquidity Consistency Validation:

```
// Add liquidity consistency checking
function _validateLiquidityConsistency() private view {
    uint256 totalSupply = totalSupply();
    uint256 wethReserves = i_wethToken.balanceOf(address(this));
    uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));

    require(totalSupply > 0 || (wethReserves == 0 && poolTokenReserves == 0),
            "Liquidity consistency violation");
}
```

Additional Measures: - Implement comprehensive liquidity monitoring - Add alerts for liquidity consistency violations - Implement automatic rebalancing mechanisms

[M-03] Unchecked WETH Token Address in Constructor

Description

The `PoolFactory` constructor accepts a `wethToken` address without validating if it's a valid ERC20 token or non-zero address.

Impact

- Contract malfunction
- Invalid token integration
- No recovery mechanism

Recommended mitigation

Add Constructor Validation:

```
constructor(address wethToken) {
    require(wethToken != address(0), "WETH token address cannot be zero");
    require(wethToken.code.length > 0, "WETH token address must be a contract");

    // Validate ERC20 interface
    try IERC20(wethToken).totalSupply() returns (uint256) {
        // Token exists
    } catch {
        revert("WETH token must implement ERC20 interface");
    }

    i_wethToken = wethToken;
}
```

Additional Safety Measures: - Implement upgradeable pattern for future WETH changes - Add governance controls for WETH token updates - Add monitoring for WETH token behavior

Informational

[I-01] Inadequate Logging and Monitoring

Description

The contract lacks comprehensive logging and monitoring capabilities, making it difficult to track suspicious activities.

Impact

- Limited audit trail
- Difficulty in detecting attacks
- Poor incident response capabilities

Recommended mitigation

Add Comprehensive Events:

```
// Add detailed events for all operations
event SwapExecuted(address indexed user, IERC20 indexed inputToken, uint256 inputAmount, IEI
event LiquidityAdded(address indexed provider, uint256 wethAmount, uint256 poolTokenAmount,
event EmergencyAction(address indexed caller, string action, uint256 timestamp);
```

Additional Monitoring Measures: - Implement real-time monitoring dashboard - Add automated alerting for suspicious activities - Implement comprehensive analytics