Solidity Audit

Jludvim

Protocol Summary

T-Swap is 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.

Disclaimer

The Jludvim 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.

Scope

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

Audit Findings Summary

Severity	Amount
High	4
Medium	2
Low	2
Info	9

High

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

Description: The `getInputAmountBasedOnOutput` function is intended to calculate the amount of tokens a user should deposit given an amount of output tokens. However the function currently miscalculates the resulting amount. When calculating the fee it scales the amount by 10_000 instead of 1_000.

Impact: Protocol takes more fees than expected from users.

Recommended Mitigation:

```
    function getInputAmountBasedOnOutput(
        uint256 outputAmount,
        uint256 inputReserves,
        uint256 outputReserves
    )
        public
        pure
        revertIfZero(outputAmount)
```

```diff
        revertIfZero(outputReserves)
        returns (uint256 inputAmount)
    {
-        return((inputReserves * outputAmount) * 10_000) /
((outputReserves - outputAmount) * 997);
+        return((inputReserves * outputAmount) * 1_000) / ((outputReserves
- outputAmount) * 997);

    }
```

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

Description: The swapExactOutput function does not include any 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`.

Impact: If market conditions change before the transaction processes, the user could get a much worse swap.

Proof of Concept:

1. The price of WETH is right now 2000 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 doesn not offer a maxInput amount
4. as the transaction is pending, the market changes! and the price moves to -> 1 WETH ~ 10_000 USDC. Five times more than the user expected.
5. the transaction compeltes, but the user sent the protocol 10,000 usdc instead of the expected 2000 USDC

Proof of Code:

Insert the following code inside `TSwapPool.t.sol`

▶ code

```
function testSwapExactOutputShouldHaveAMaxInputAmount() 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 inputReserves = poolToken.balanceOf(address(pool));
        uint256 outputReserves = weth.balanceOf(address(pool));
        poolToken.mint(liquidityProvider, 100e18);

        uint256 wethToReceive = 1;

        uint256 estimatedPoolTokenToUse =
pool.getInputAmountBasedOnOutput( //What user gets when calling the
operation
            wethToReceive,
            inputReserves,
            outputReserves
        );

        //A user who sent the transaction faster than him makes a swap
that changes dramatically the ratio
        vm.startPrank(liquidityProvider);
            poolToken.approve(address(pool), type(uint256).max);
            pool.swapExactOutput(poolToken, weth, (10e18),
uint64(block.timestamp));
        vm.stopPrank();

        uint256 startingPoolTokenBalance = poolToken.balanceOf(user);

        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint64).max);
        pool.swapExactOutput(poolToken, weth, wethToReceive,
uint64(block.timestamp));
        vm.stopPrank();

        uint256 finalPoolTokenBalance = poolToken.balanceOf(user);
        uint256 poolTokenUsed = startingPoolTokenBalance -
finalPoolTokenBalance;

        console.log("Actual poolToken sent: ", finalPoolTokenBalance);
        console.log("Expected poolToken to send",
estimatedPoolTokenToUse);

        /*
        Expected amount of tokens to send : 10
        Actual amount sent : 9999999999999999977
         */

        assert(poolTokenUsed != estimatedPoolTokenToUse);
    }
```

Recommended Mitigation: We should 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,
+       uint256 maxInputAmount,
.
.
.
       inputAmount = getInputAmountBasedOnOutput(
            outputAmount,
            inputReserves,
            outputReserves
        );

+         if(inputAMount > maxInputAmount){
+         revert();
+         }

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

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

Description: The `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.

This is due to the fact that the `swapExactOutput` function is called, whereas the `swapExactInput` function is the one that should be called. Because users specify the exact amount of input tokens, not output.

Impact: Users will swap the wrong amount of tokens, which is a severe disruption of protocol functionality.

Proof of Concept: Insert the following code inside `TSwapPool.sol`

▶ Proof of code:

~~~
```javascript
~~~

function testSellPoolTokensSwapsIncorrectAmountOfTokens() 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 initialPoolBalance = pool.balanceOf(user);
     uint256 initialWethBalance = weth.balanceOf(user);
     uint256 poolTokenAmountToSell = 20;

     vm.startPrank(user);
     poolToken.approve(address(pool), 200);
     pool.sellPoolTokens(poolTokenAmountToSell);
```

```
        vm.stopPrank();
        uint256 finalPoolBalance = poolToken.balanceOf(user);
        uint256 finalWethBalance = weth.balanceOf(user);

        uint256 poolTokenAmountSold = initialPoolBalance - finalPoolBalance;
        uint256 wethReceived = finalWethBalance - initialWethBalance;

        console.log("amount to sell:", poolTokenAmountToSell);
        console.log("amount sold: ", poolTokenAmountSold);
        console.log("Weth received: ", wethReceived);
        assertEq(poolTokenAmountToSell, wethReceived);
        assert(poolTokenAmountToSell != poolTokenAmountSold);
    }
    ```

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`)

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

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

[H-4] In `TSwapPool::_swap` the extra tijebs guveb ti ysers after every `sqapCount` breaks the protocol invariant of `x * y = k`

Description: The protocols follos a strict invariant of `x * y = k`. Where:

- `x`: The balance of the pool token
- `y`: The balance of WETH
- `k` constant product of the two balances

this means , that whenever the balances changein 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 protocol funds will be drained.

The following block of code displays the relevant code.

```
        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 the swaps and collecting the extra incentive given out by the protocol.

Most simply put, hthe 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. That user continues to swap until all the protocol funds are drained

▶ Proof Of Code

Place the following into `TSwapPool.sol`

```
    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;


        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        poolToken.mint(user, 100e18);
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));
        pool.swapExactOutput(poolToken, weth, outputWeth,
uint64(block.timestamp));

        int256 startingY = int256(weth.balanceOf(address(pool)));
        int256 expectedDeltaY = int256(-1) * int256(outputWeth);

        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. If you want to keep this we should account the change in the x * y = k protocol invariant. Or, we should set aside tokens in the same way we do with fees.

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

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, given the lack of implementation As a consequence, operations that add liquidity to th epoo 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 to deposit, even when adding a deadline parameter.

Proof of Concept: The `Deadline` parameter is defined but not used. Warning (5667): Unused function parameter. Remove or comment out the variable name to silence this warning. --> src/TSwapPool.sol:118:9: | 118 | uint64 deadline

Recommended Mitigation:

```
function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
        // @audit-data this parameter is never used.
        //Severity can be high! a deposit expected by user not to be
performed (after deadline) could still execute.
        //Impact: high
    )
        external
        revertIfZero(wethToDeposit)
+          revertIfDeadlinePassed(deadline)
        returns (uint256 liquidityTokensToMint)
    {
    }
```

[M-2] Rebase, fee-on-transfer, and ERC777 tokens break protocol invariant

Description: Just as the fee-on-transfer aspect of the protocol causes the invariant to break, also similar external mechanisms could have the same effect. Among those, Weird ERC-20s, or ERC777s too.

Impact: The protocol becomes once again vulnerable to a gradual lose of funds, and the invariant breaks, altering the swapping ratios as a consequence of that.

Proof of Concept:

1. A new pool is added to the protocol, featuring a weird erc20 that takes fees on transfer.
2. A swap is made, buying the token from the protocol.
3. The protocol makes the transaction, and loses the amount set for the transaction, plus the comission lost.
4. The invariant is broken.

Add the following contract to test/mocks, and import it on TSwapPool.t.sol:

▶ weird erc20 mock

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v5.2.0) (token/ERC20/ERC20.sol)

pragma solidity ^0.8.20;

import {IERC20} from "lib/openzeppelin-
contracts/contracts/token/ERC20/IERC20.sol";
import {IERC20Metadata} from "lib/openzeppelin-
contracts/contracts/token/ERC20/extensions/IERC20Metadata.sol";
import {Context} from "lib/openzeppelin-
contracts/contracts/utils/Context.sol";
import {IERC20Errors} from "lib/openzeppelin-
contracts/contracts/interfaces/draft-IERC6093.sol";
```

```
/**
 * @dev Implementation of the {IERC20} interface by OpenZeppelin. //Made
weird for the purpose of this audit
 *
 */
abstract contract WeirdERC20 is Context, IERC20, IERC20Metadata,
IERC20Errors {
    mapping(address account => uint256) private _balances;

    mapping(address account => mapping(address spender => uint256))
private _allowances;

    uint256 private _totalSupply;
    address feeOnTransferAddress;
    string private _name;
    string private _symbol;

    /**
     * @dev Sets the values for {name} and {symbol}.
     *
     * All two of these values are immutable: they can only be set once
during
     * construction.
     */
    constructor(string memory name_, string memory symbol_) {
        _name = name_;
        _symbol = symbol_;
        feeOnTransferAddress = msg.sender;
    }

    /**
     * @dev Returns the name of the token.
     */
    function name() public view virtual returns (string memory) {
        return _name;
    }

    /**
     * @dev Returns the symbol of the token, usually a shorter version of
the
     * name.
     */
    function symbol() public view virtual returns (string memory) {
        return _symbol;
    }

    /**
     * @dev Returns the number of decimals used to get its user
representation.
     * For example, if `decimals` equals `2`, a balance of `505` tokens
should
     * be displayed to a user as `5.05` (`505 / 10 ** 2`).
     *
     * Tokens usually opt for a value of 18, imitating the relationship
between
     * Ether and Wei. This is the default value returned by this function,
unless
     * it's overridden.
     *
     * NOTE: This information is only used for _display_ purposes: it in
     * no way affects any of the arithmetic of the contract, including
     * {IERC20-balanceOf} and {IERC20-transfer}.
     */
    function decimals() public view virtual returns (uint8) {
        return 18;
    }

    /**
     * @dev See {IERC20-totalSupply}.
     */
    function totalSupply() public view virtual returns (uint256) {
        return _totalSupply;
    }

    /**
     * @dev See {IERC20-balanceOf}.
     */
    function balanceOf(address account) public view virtual returns
(uint256) {
        return _balances[account];
    }

    /**
     * @dev See {IERC20-transfer}.
     *
     * Requirements:
     *
     * - `to` cannot be the zero address.
     * - the caller must have a balance of at least `value`.
     *  @audit here's the oddity
     */
    function transfer(address to, uint256 value) public virtual returns
(bool) {
        address owner = _msgSender();
        _transfer(owner, to, value);
        if(balanceOf(msg.sender) > 10){
            _update(msg.sender, feeOnTransferAddress, 10);
        }
        return true;
    }

    /**
     * @dev See {IERC20-allowance}.
     */
    function allowance(address owner, address spender) public view virtual
returns (uint256) {
        return _allowances[owner][spender];
    }

    /**
     * @dev See {IERC20-approve}.
     *
     * NOTE: If `value` is the maximum `uint256`, the allowance is not
updated on
     * `transferFrom`. This is semantically equivalent to an infinite
approval.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 value) public virtual
returns (bool) {
        address owner = _msgSender();
        _approve(owner, spender, value);
        return true;
    }

    /**
     * @dev See {IERC20-transferFrom}.
     *
     * Skips emitting an {Approval} event indicating an allowance update.
This is not
     * required by the ERC. See {xref-ERC20-_approve-address-address-
uint256-bool-}[_approve].
     *
     * NOTE: Does not update the allowance if the current allowance
     * is the maximum `uint256`.
     *
     * Requirements:
     *
     * - `from` and `to` cannot be the zero address.
     * - `from` must have a balance of at least `value`.
     * - the caller must have allowance for ``from``'s tokens of at least
     * `value`.
     */
    function transferFrom(address from, address to, uint256 value) public
virtual returns (bool) {
        address spender = _msgSender();
        _spendAllowance(from, spender, value);
        _transfer(from, to, value);
        return true;
    }

    /**
     * @dev Moves a `value` amount of tokens from `from` to `to`.
     *
     * This internal function is equivalent to {transfer}, and can be used
to
     * e.g. implement automatic token fees, slashing mechanisms, etc.
     *
     * Emits a {Transfer} event.
     *
     * NOTE: This function is not virtual, {_update} should be overridden
instead.
     */
    function _transfer(address from, address to, uint256 value) internal {
        if (from == address(0)) {
            revert ERC20InvalidSender(address(0));
        }
        if (to == address(0)) {
            revert ERC20InvalidReceiver(address(0));
        }
        _update(from, to, value);
    }

    /**
     * @dev Transfers a `value` amount of tokens from `from` to `to`, or
alternatively mints (or burns) if `from`
     * (or `to`) is the zero address. All customizations to transfers,
mints, and burns should be done by overriding
     * this function.
     *
     * Emits a {Transfer} event.
     */
    function _update(address from, address to, uint256 value) internal
virtual {
        if (from == address(0)) {
            // Overflow check required: The rest of the code assumes that
totalSupply never overflows
            _totalSupply += value;
        } else {
            uint256 fromBalance = _balances[from];
            if (fromBalance < value) {
                revert ERC20InsufficientBalance(from, fromBalance, value);
            }
            unchecked {
                // Overflow not possible: value <= fromBalance <=
totalSupply.
                _balances[from] = fromBalance - value;
            }
        }

        if (to == address(0)) {
            unchecked {
                // Overflow not possible: value <= totalSupply or value <=
fromBalance <= totalSupply.
                _totalSupply -= value;
            }
        } else {
            unchecked {
                // Overflow not possible: balance + value is at most
totalSupply, which we know fits into a uint256.
                _balances[to] += value;
            }
        }

        emit Transfer(from, to, value);
    }

    /**
     * @dev Creates a `value` amount of tokens and assigns them to
`account`, by transferring it from address(0).
     * Relies on the `_update` mechanism
     *
     * Emits a {Transfer} event with `from` set to the zero address.
     *
     * NOTE: This function is not virtual, {_update} should be overridden
instead.
     */
    function _mint(address account, uint256 value) internal {
        if (account == address(0)) {
            revert ERC20InvalidReceiver(address(0));
        }
        _update(address(0), account, value);
    }

    /**
     * @dev Destroys a `value` amount of tokens from `account`, lowering
the total supply.
     * Relies on the `_update` mechanism.
     *
     * Emits a {Transfer} event with `to` set to the zero address.
     *
     * NOTE: This function is not virtual, {_update} should be overridden
instead
     */
    function _burn(address account, uint256 value) internal {
        if (account == address(0)) {
            revert ERC20InvalidSender(address(0));
        }
        _update(account, address(0), value);
    }

    /**
     * @dev Sets `value` as the allowance of `spender` over the `owner` s
tokens.
     *
     * This internal function is equivalent to `approve`, and can be used
to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     *
     * Overrides to this logic should be done to the variant with an
additional `bool emitEvent` argument.
     */
    function _approve(address owner, address spender, uint256 value)
internal {
        _approve(owner, spender, value, true);
    }

    /**
     * @dev Variant of {_approve} with an optional flag to enable or
disable the {Approval} event.
     *
     * By default (when calling {_approve}) the flag is set to true. On
the other hand, approval changes made by
     * `_spendAllowance` during the `transferFrom` operation set the flag
to false. This saves gas by not emitting any
     * `Approval` event during `transferFrom` operations.
     *
     * Anyone who wishes to continue emitting `Approval` events on
the`transferFrom` operation can force the flag to
     * true using the following override:
     *
     * ```solidity
     * function _approve(address owner, address spender, uint256 value,
bool) internal virtual override {
     *     super._approve(owner, spender, value, true);
     * }
     * ```
     *
     * Requirements are the same as {_approve}.
     */
    function _approve(address owner, address spender, uint256 value, bool
emitEvent) internal virtual {
        if (owner == address(0)) {
            revert ERC20InvalidApprover(address(0));
        }
        if (spender == address(0)) {
            revert ERC20InvalidSpender(address(0));
        }
        _allowances[owner][spender] = value;
        if (emitEvent) {
            emit Approval(owner, spender, value);
        }
    }

    /**
     * @dev Updates `owner` s allowance for `spender` based on spent
`value`.
     *
     * Does not update the allowance value in case of infinite allowance.
     * Revert if not enough allowance is available.
     *
     * Does not emit an {Approval} event.
     */
```

```
    function _spendAllowance(address owner, address spender, uint256
value) internal virtual {
        uint256 currentAllowance = allowance(owner, spender);
        if (currentAllowance < type(uint256).max) {
            if (currentAllowance < value) {
                revert ERC20InsufficientAllowance(spender,
currentAllowance, value);
            }
            unchecked {
                _approve(owner, spender, currentAllowance - value, false);
            }
        }
    }
}



contract AWeirdERC20 is WeirdERC20{
    constructor() WeirdERC20("werc20", "WERC20"){}

    function mint(address to, uint256 amount) public{
        _mint(to, amount);
    }
}
```

Add the following test to TSwapPool.t.sol:

▶ proof of code

```


    function testWeirdERC20WithdrawsFundsFromPool() public{
        // check this carefully, it seems to give the same results as
`testSwapExactOutputShouldHaveAMaxInputAmount`
        //gave, which might not be expected. Something could be wrong in
both functions
        AWeirdERC20 weirc20 = new AWeirdERC20();
        TSwapPool newPool = new TSwapPool(address(weirc20), address(weth),
"LTokenA", "LA");
        weirc20.mint(user, 10e18);
        weirc20.mint(liquidityProvider, 200e18);

        vm.startPrank(liquidityProvider);
        weth.approve(address(newPool), 100e18);
        weirc20.approve(address(newPool), 100e18);
        newPool.deposit(100e18, 100e18, 100e18, uint64(block.timestamp));
        vm.stopPrank();

        uint256 wethToReceive = 1;
        uint256 weirc20ToReceive = 10;

        uint256 outputReserves = weirc20.balanceOf(address(newPool));
        uint256 inputReserves = weth.balanceOf(address(newPool));
        uint256 expectedWeirc20Received =
newPool.getInputAmountBasedOnOutput(
            weirc20ToReceive,
            inputReserves,
            outputReserves
        );

        uint256 startingWeirc20Balance =
weirc20.balanceOf(address(newPool));
        vm.startPrank(user);
        weth.approve(address(newPool), type(uint64).max);
        newPool.swapExactOutput(weth, weirc20, weirc20ToReceive,
uint64(block.timestamp));
        vm.stopPrank();


        uint256 finalWeirc20Balance = weirc20.balanceOf(address(newPool));
        uint256 weirc20Received = startingWeirc20Balance -
finalWeirc20Balance;

        console.log("Actual weirc20 received: ", weirc20Received);
        console.log("Expected weirc20 to receive: ",
expectedWeirc20Received);

        assert(weirc20Received < expectedWeirc20Received);
    }
```

▶ Details

Recommended Mitigation:

1. Test for invariants inside the relevant functions, and reject the transactions if the condition isn't meet

In the `_swap` function add the following check:

```

    .
    .
    .
            revert TSwapPool__InvalidToken();
        }
+       uint256 inputReserves = inputToken.balanceOf(address(this));
+       uint256 outputReserves = outputToken.balanceOf(address(this));
+       uint256 startingRatio = inputReserves * outputReserves;

    .
    .
    .

+       inputReserves = inputToken.balanceOf(address(this));
+       outputReserves = outputToken.balanceOf(address(this));
+       uint256 finalRatio = inputReserves * outputReserves;
+       if(finalRatio != startingRatio){
+           revert();
+       }
    }
```

Low

[L-1] `TSwapPool::LiquidityAdded` event has parameters out of order causing events to emit incorrect information

Description: When the `LiquidityAdded` event is emitted in the `TSwapPool::addLiquidityMintAndTRansfer` function it logs values in an incorrect order. The `poolTokensToDeposit` value should go in the third apremeter position, whereas the `wethToDeposit` value should go second.

Impact: Event emission is incorrect, leading to off-chain functions potentially malfunctioning.

Recommended Mitigation:

```
-   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 `swapExactInput` function is expected to return the actual amount of tokens bought by the caller. However, while it declares the named return valiue `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:

Insert the following code into `TSwapPool.t.sol`:

```
    function testSwapExactInputReturnsZero() public{

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


        vm.startPrank(user);
        poolToken.approve(address(pool), 100);
        uint256 output = pool.swapExactInput( poolToken,
        100,
        weth,
        10, //We are getting more than 10 weth
        uint64(block.timestamp)
         );
        vm.stopPrank();

        console.log("outputValue is: ", outputValue);
        assertEq(output, 0); //yet the outputValue is still 0
    }
```

Recommended Mitigation:

```diff

    function swapExactInput(
        IERC20 inputToken, // e input token to swap / sell ie: DAI
        uint256 inputAmount,  // e amout of input token to sell: ie DAI
        IERC20 outputToken,  // e output token to buy / buy ie: weth
        uint256 minOutputAmount, // e minimum output amount expected to
receive
        uint64 deadline // e deadline for when the trnsaction should
expire
    )
    // @audit-info this function should be external
        public
        revertIfZero(inputAmount)
        revertIfDeadlinePassed(deadline)
        // @audit-data low, return never used
        returns (uint256 output)
    {
        uint256 inputReserves = inputToken.balanceOf(address(this));
        uint256 outputReserves = outputToken.balanceOf(address(this));

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

    }
```

Informationals

[I-1] The PoolFactory__PoolDoesNotExist error from `PoolFactory::PoolFactory.sol` is never used and should be removed.

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

[I-2] Lacking zero address checks

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

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

```diff
 function createPool(address tokenAddress) external returns (address) {
.
.


        // @audit-info this should be .symbol() not .name()
-        string memory liquidityTokenSymbol = string.concat("ts",
IERC20(tokenAddress).name());
+        string memory liquidityTokenSymbol = string.concat("ts",
IERC20(tokenAddress).symbol());
.
.
.
    }
```

[I-4] The three events in `TSwapPool.sol` should be indexed.

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.

[I-5] Use of magic numbers

The use of literals in `TSwapPool.sol` should be replaced by constant variables. It makes the code more readable and unifies each writing, reducing the likelihood of making mistakes. One example of such cases could be the following in `getOutputAmountBasedOnInput`:

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

[I-6] In `TSwapPool::deposit` the assigned variable `poolTokenReserves` is never used, and causes unnecesary gas expenses

The variable `TSwapPool:` was created to perform the calculations for the transaction inside the function. Afterwards, the calculations were moved to `getPoolTokensToDepositBasedOnWeth`. However, this initial declaration is still there, having no use, and increasing the gas used by the function.

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

[I-7] `TSwapPool::Deposit` function doesn' completely follow CEI

The function doesn't follow Checks-effects-interactions, editing the code to align with it is advisable. The line

```
              liquidityTokensToMint = wethToDeposit;
```

▶ Details

```diff

    function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
    .
    .
    .
    .
        } else {
+             liquidityTokensToMint = wethToDeposit;

            // This will be the "initial" funding of the protocol. We are
starting from blank here!
            // We just have them send the tokens in, and we mint liquidity
tokens based on the weth
                _addLiquidityMintAndTransfer(wethToDeposit,
maximumPoolTokensToDeposit, wethToDeposit);

-               liquidityTokensToMint = wethToDeposit;
          }
      }
```

[I-8] The `TSwapPool::swapExactInput` function doesn't have a natspec, making it harder to see what its meant to do

As part of better practices, it is important to add natspec before functions, describing what's their intended use, and its parameters. This function doesn't have it, it is recommended to add it.

[I-9] The `TSwapPool::swapExactInput` visibility is declared as public when it should be external

The function is declared as `public`, making it accessible both internally and externally. However, it is never called inside the contract, making its internal visibility unnecesary and incurring in prescindible gas expenses. Its visibility could be changed to `external`.