High

[H-1] `TSwapPool::deposit` is missing deadline check causing transaction to complete even after the deadline

Description:

The `deposit` function accept the deadline parameter, which according to the documentation is `/// @param deadline The deadline for the transaction to be completed by`.However this parameter is never used.As a consequence , operations that add liquidity pool might be executed at unexpected times, in market conditions where the deposit rate is unfavourable.

Impact:

Transaction could be sent when market conditions are unfavourable to deposit, even when adding a deadline paramerter.

Proof of Concept:

The `deadline` parameter is unused.

Recommended Mitigation:

Consider making the fallowing change to the function.

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

[H-2] Incorrect fee calculation in `TSwapPool::getInputAmountBasedOnOutput`

causes protocal to take too 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 amount of tokens of output tokens. However, the function currently misscalculats the resultant amount.When calculating the fee, it scales the amount by 10_000 instead of 1_000.

Impact:

Protocal takes more fees then expected from user.

Recommended Mitigation:

```
    function getInputAmountBasedOnOutput(
    uint256 outputAmount,
    uint256 inputReserves,
    uint256 outputReserves
)
    public
    pure
    revertIfZero(outputAmount)
    revertIfZero(outputReserves)
    returns (uint256 inputAmount)
{
    return
-        ((inputReserves * outputAmount) * 10000) /
        ((outputReserves - outputAmount) * 997);

+        ((inputReserves * outputAmount) * 1000) /
        ((outputReserves - outputAmount) * 997);
}
```

[H-3] Lack of slippage protection `TSwapPool::swapExactOutput` causes users to potentially receive way fewer tokens

Description:

The `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 the `minOutputAmount`, the `swapExactOutput` function should specify a `maxInputAmount`.

Impact:

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

Proof of Concept:

1. The price of WETH right now is 1_000 USDC.
2. User input the `swapExactOutput` looking for 1 WETH
    1. inputToken = USDC
    2. outputToken = WETH
    3. outputAmount = 1
    4. deadline = whatever
3. The function does not offfer a maxInput amount.
4. As the transaction is pending in the mempool, the market changes! And the price moves hugh --> 1 WETH is now 10_000 USDC. 10x more then the user expected.
5. The transaction completes, but the user sent the protocol 10_000 USDC instead of expected 1_000 USDC

Recommended Mitigation:

We should include a `maxInputAmount` so the user only has to spend up to a specific amount, and can predict how they will spend on protocol.

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

     inputAmount = getInputAmountBasedOnOutput(
         outputAmount,
         inputReserves,
         outputReserves
```

```
        );

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

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

[H-4] The `TSwapPool::sellPoolTokens` mismatches input and output tokens causing user 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 amny pool tokens they are willing to sell in the `poolTokenAmount` parameter. However the function miscalculate the swapped amount.

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

Impact:

USer will swap the wrong amount of tokens, which is the servere disruption of protocol functionality.

Recommended Mitigation:

Consider the implementation to use `swapExactInput` instead of use `swapExactOutput`. Note that this would also require changing the `sellPoolTokens` function to accept a new parameter (ie minWethToReceive[minOutputAmount]) to be passed to `swapExactInput`.

```
      function sellPoolTokens(
        uint256 poolTokenAmount,
    +   uint256 minWethToReceive,
    +   uint64 deadline
    ) 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)
+       );

}
```

[H-5] In `TSwapPool::_swap` the extra tokens given to users after every `swapCount` breaks the protocol invariant of `x * y = k`.

Description:

The protocol fallow a strict invariant of `x * y = k`. Where:

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

  This means, that whenever the balance change in the protocol, the ratio between the two amount 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 fallow 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 swap 10 times and collect, and collect the extra incentive pf `1_000_000_000_000_000_000` tokens.
2. A user continous to swap until all the protocol funds are drained.

▶ Proof Of Code

Place the fallowing into `TSwapPool.t.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 = 1e18;
        vm.startPrank(user);
        poolToken.approve(address(pool), type(uint256).max);
        poolToken.mint(user, 1000e18);
        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();

        int256 endingY = int256(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 in, we should account for the chnage in x * y = k protocol invariant. Or, we should set aside tokens in the same way we do with fees.

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

Low

[L-1] `TSwapPool::LiquidityAdded` event has parameters out of order

Description:

When the `LiquidityAdded` event is emitted in the `TSwapPool::_addLiquidityMintAndTransfer` function , is log values in the incorrect order. The `poolTokensToDeposit` value should go in their 3rd parameter position, whereas the `wethToDeposit` value should go in their 2nd position.

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] Default value returned by `TSwapPool::swapExactInput` result 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 value `output` it is never assigned a value, nor uses an explict return statement.

Impact:

The return value will always be zero, giving incorrect information to the caller.

Recommended Mitigation:

```
        function swapExactInput(
         IERC20 inputToken,
         uint256 inputAmount,
         IERC20 outputToken,
         uint256 minOutputAmount,
         uint64 deadline
    )
         public
         revertIfZero(inputAmount)
         revertIfDeadlinePassed(deadline)
         /// @audit in return output is unused , always return 0
         /// instead of output --> outputAmount in returns
         returns (uint256 outputAmount)
    {
          uint256 inputReserves = inputToken.balanceOf(address(this));
         uint256 outputReserves = outputToken.balanceOf(address(this));

         outputAmount = getOutputAmountBasedOnInput(
             inputAmount,
             inputReserves,
             outputReserves
         );

         if (outputAmount < minOutputAmount) {
             revert TSwapPool__OutputTooLow(outputAmount, minOutputAmount);
         }

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

```
    }
```

Informationals

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

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

[I-2] Lacking zero address check

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

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

```
event Swap(
    address indexed swapper,
```

```
        IERC20 tokenIn,
        uint256 amountTokenIn,
        IERC20 tokenOut,
        uint256 amountTokenOut
    );
```