Protocol Audit Report

Version 1.0

Cyfrin.io

October 16, 2024

TSwap Audit Report

CC

October 16, 2024

Prepared by: CasinoCompiler

Lead Auditors: CC

Table of Contents

Protocol Summary

The protocol is a copy of Uniswap V1, having all of the main functionality; A creation contract that allows for pools to be created between any ERC20 token and WETH. These pools allow for liquidity to be provided and withdrawn, and the constitution of the pool is denoted through the use of liquidity tokens. None-liquidity providers can swap between the ERC20 and WETH whenever they wish to provided there's enough liquidity in the pool. The protocol accumulates fees per trade and all fees are added to the pool.

Disclaimer

The CC 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
Likelihood	High	H	H/M	M
	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 to the following commit hash:

```
1ec3c30253423eb4199827f59cf564cc575b46db
```

Scope

```
./src/
==> PoolFactory.sol
==> TSwapPool.sol
```

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

In total, 13 security vulnerabilities were found, 7 of which tampered with the integrity of the protocol.

Issues found

Severity	Number of Issues Found
High	4
Medium	1
Low	2
Informational	5
Gas	1
Total	13

Findings

High

[H-1] Core Invariant of prot0col broken due to swap incentive in internal function `TSwapPool.sol::_swap()`

Description:

The protocol follows a strict invariant of `x * y = k`. Where: - `x`: The balance of the pool token - `y`: The balance of WETH - `k`: 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 protocol funds will be drained.

The follow block of code is responsible for the issue.

```
1           swap_count++;
2           if (swap_count >= SWAP_COUNT_MAX) {
3               swap_count = 0;
4               outputToken.safeTransfer(msg.sender, 1
                    _000_000_000_000_000_000);
5           }
```

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. That user continues to swap untill all the protocol funds are drained

Proof Of Code

Place the following 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 = 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();

```

```
30          uint256 endingY = weth.balanceOf(address(pool));
31          int256 actualDeltaY = int256(endingY) - int256(startingY);
32          assertEq(actualDeltaY, expectedDeltaY);
33      }
```

Recommended Mitigation:

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

```
1 -         swap_count++;
2 -         // Fee-on-transfer
3 -         if (swap_count >= SWAP_COUNT_MAX) {
4 -             swap_count = 0;
5 -             outputToken.safeTransfer(msg.sender, 1
      _000_000_000_000_000_000);
6 -         }
```

[H-2] Math precision is incorrect in `TSwapPool.sol::getInputAmountBasedOnOutput()` leading to incorrect fee calculation.

Description:

The `getInputAmountBasedOnOutput` function is inteded to calculate the amount of tokens a user should deposit given an amount of output tokens. The function is currently miscalculating the the amount as the amount of fees being calculated is incorrect; it scales the fees by 10_000 instead of 1_000.

Impact:

protocol takes more fees than expected from users.

Proof of Concept:

1.
   - Assume initial liquidity has been provided for 100e18 weth and 100e18 pool tokens.
   - Assume a user wants to swap 1e18 weth equivalent.
2. We expect the expected return to be calculated as such: $(inputReserves * outputAmount) * 1000)/((outputReserves - outputAmount) * 997)$
3. Insert the following test into foundry:

```
1     function test_High_2() public provideInitialLiquidity {
2         uint256 expectedInputReturn = ((poolToken.balanceOf(address(
              pool)) * 1e18) * 1000) / ((weth.balanceOf(address(pool)) - 1
              e18) * 997);
3 
4         vm.startPrank(user);
```

```
5           uint256 inputReturn = pool.getInputAmountBasedOnOutput(1e18,
                poolToken.balanceOf(address(pool)), weth.balanceOf(address(
                pool)));
6
7           assert(expectedInputReturn != inputReturn);
8       }
```

4. The assertion will fail as the expected return (input required)is 1_013_140_431_395_195_688 but the actual input return is 10_131_404_313_951_956_880

Recommended Mitigation:

```
1       function getInputAmountBasedOnOutput(
2           uint256 outputAmount,
3           uint256 inputReserves,
4           uint256 outputReserves
5       )
6           public
7           pure
8           revertIfZero(outputAmount)
9           revertIfZero(outputReserves)
10          returns (uint256 inputAmount)
11      {
12  -       return ((inputReserves * outputAmount) * 10000) / ((
        outputReserves - outputAmount) * 997);
13  +       return ((inputReserves * outputAmount) * 1_000) / ((
        outputReserves - outputAmount) * 997);
14      }
```

[H-3] No slippage protection in `TSwapPool.sol::swapExactOutput()` causing users to potentially receive a lot less 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 a `minOutputAmount`, the `swapExactOutput` function should specify a `maxInputAmount`.

Impact:

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

Proof of Concept: 1. The price of 1 WETH right now is 1,000 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 10,000 USDC. 10x more

than the user expected 5. The transaction completes, but the user sent the protocol 10,000 USDC instead of the 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 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-4] `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 protcol functionality.

Proof of Concept:

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,
```

```
4           ) external returns (uint256 wethAmount) {
5   -           return swapExactOutput(i_poolToken, i_wethToken,
        poolTokenAmount, uint64(block.timestamp));
6   +           return swapExactInput(i_poolToken, poolTokenAmount,
        i_wethToken, minWethToReceive, uint64(block.timestamp));
7       }
```

Medium

[M-1] `deadline` isn't being checked for in `TSwapPool.sol::deposit()`

Description:

The input parameter `uint64 deadline` in `TSwapPool.sol::deposit()` hasn't been used as intended as no check is in place to see if the dealine has not been passed.

Impact:

Liquidity providers may have their transactions executed at unexpected times.

Proof of Concept:

Unused `uint64 dealine` input parameter.

Recommended Mitigation:

Implement deadline check through `TSwapPool.sol::revertIfDeadlinePassed()` modifier in the same manner as it has been implemented in `TSwapPool.sol::swapExactInput()` / `TSwapPool.sol::swapExactOutput()`.

Code

```
1
2       function deposit(
3           uint256 wethToDeposit,
4           uint256 minimumLiquidityTokensToMint,
5           uint256 maximumPoolTokensToDeposit,
6           // @audit - High:: deadline isn't being checked with
                revertIfDeadlinePassed()
7           uint64 deadline
8       )
9           external
10          revertIfZero(wethToDeposit)
11  +       revertIfDeadlinePassed(deadline)
12          returns (uint256 liquidityTokensToMint)
13      {
14          // ... Rest of code
15      }
```

Low

[L-1] Order of parameters in emission `LiquidityAdded` in internal function `TSwapPool.sol::_addLiquidityMintAndTransfer()` is incorrect.

Description:

The order of the parameters in the emission of the `LiquidityAdded` event is incorrect. The correct event is:

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

Impact:

Event emissions are incorrect therefore off-chain functions that rely on events would not function as intended.

Recommended Mitigation:

code

```
    function _addLiquidityMintAndTransfer(
        uint256 wethToDeposit,
        uint256 poolTokensToDeposit,
        uint256 liquidityTokensToMint
    )
        private
    {
        _mint(msg.sender, liquidityTokensToMint);
-       emit LiquidityAdded(msg.sender, poolTokensToDeposit, wethToDeposit);
+       emit LiquidityAdded(msg.sender,  wethToDeposit, poolTokensToDeposit);
        // ...
    }
```

[L-2] Default value returned for `TSwapPool.sol::swapExactInput()` results in incorrect value returned.

Description:

`swapExactInput()` has a return value of `output` which infers to a user the output are getting based on input. The value is never assigned during function execution therefore the return value will always be 0.

Impact:

Users may read the output as 0 and be lead to believe they never received an output.

Proof of Concept:

1. Initial liquidity has been provided by LiquidityProvider and User has given approval to pool contract.
2. Run the following test in foundry.

```
    function test_Low_2() public provideInitialLiquidity
       userGivesApproval {
        vm.startPrank(user);
        uint256 outputReturn = pool.swapExactInput(poolToken, 1e18,
           weth, 1e9, uint64(block.timestamp));

        assert(outputReturn == 0);
    }
```

3. the expected output would be 1e9, however, the assertion asserts that the output is 0.

Recommended Mitigation:

Modify `output` variable name to `outputAmount` as this would follow original code and give intended outcome.

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

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

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

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

Gas

[G-1] Unused `poolTokenReserves` parameter in `TSwapPool.sol::deposit()`

Description:

`poolTokenReserves` is an unused variable.

Impact:

Increase contract deployment gas and can also cause confusion in future review.

Recommended Mitigation:

code

```
    function deposit(
        uint256 wethToDeposit,
        uint256 minimumLiquidityTokensToMint,
        uint256 maximumPoolTokensToDeposit,
        // @audit - High:: deadline isn't being checked with revertIfDeadlinePassed()
        uint64 deadline
    )
        external
        revertIfZero(wethToDeposit)
        returns (uint256 liquidityTokensToMint)
    {
        if (wethToDeposit < MINIMUM_WETH_LIQUIDITY) {
            revert TSwapPool__WethDepositAmountTooLow(MINIMUM_WETH_LIQUIDITY, wethToDeposit);
        }
        if (totalLiquidityTokenSupply() > 0) {
            uint256 wethReserves = i_wethToken.balanceOf(address(this));
            // @audit - gas:: unused variable
-           uint256 poolTokenReserves = i_poolToken.balanceOf(address(this));
```

Informational

[I-1] Use of magic numbers

Description:

The usage of magic numbers in these places:

1. `TSwapPool.sol::getOutputAmountBasedOnInput` lines 276 278

2. `TSwapPool.sol::getinputAmountBasedOnOutput` lines 294 295

Impact:

Bad code practice that can lead to math logic or precision errors, additionally makes it harder for retrospective review.

Recommended Mitigation:

Create constant variables with easily identifiable names.

[I-2] No natspec given for core functions

Description:

No natspec given for the follwing core functions:

1. `TSwapPool.sol::swapExactInput`
2. `TSwapPool.sol::swapExactOutput`

Impact:

Makes it difficult for personal, third person or retrospective review.

Recommended Mitigation:

Add natspec.

[I-3] unused error in `PoolFactory.sol`

Description:

`error PoolFactory__PoolDoesNotExist` is an unsed error.

Recommended Mitigation:

Remove custom error.

[I-4] address(0) not checked for in `PoolFactory.sol` constructor.

Recommended Mitigation:

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

[I-5] incorrect liquidity token symbol creation in `PoolFactory.sol::createPool()`

Description:

`PoolFactory.sol::createPool()` line 52 creates the liquidity pool token by concatinating 'ts' to the `token.name()` rather than the `token.symbol()`.

Recommended Mitigation:

```
1      function createPool(address tokenAddress) external returns (address
           ) {
2          if (s_pools[tokenAddress] != address(0)) {
3              revert PoolFactory__PoolAlreadyExists(tokenAddress);
4          }
5          string memory liquidityTokenName = string.concat("T-Swap ",
               IERC20(tokenAddress).name());
6  -       string memory liquidityTokenSymbol = string.concat("ts", IERC20
       (tokenAddress).name());
7  +       string memory liquidityTokenSymbol = string.concat("ts", IERC20
       (tokenAddress).symbol());
8          TSwapPool tPool = new TSwapPool(tokenAddress, i_wethToken,
               liquidityTokenName, liquidityTokenSymbol);
9          s_pools[tokenAddress] = address(tPool);
10         s_tokens[address(tPool)] = tokenAddress;
11         emit PoolCreated(tokenAddress, address(tPool));
12         return address(tPool);
13     }
```