28.03.2024

Introduction

A time-boxed security review of the first-flight **T-Swap** protocol was done by **hashov**, focused on the security aspects of the application's implementation.

Disclaimer

A smart contract security review cannot ensure the total elimination of vulnerabilities. This process is limited by time, resources, and expertise, as I strive to identify as many vulnerabilities as I can for the given timeframe. I cannot promise 100% absolute security post-review or guarantee the discovery of any issues in your smart contracts. It is highly advised to conduct additional security reviews, implement bug bounty programs, and engage in on-chain monitoring for enhanced protection.

About hashov

is a first-flight dedicated individual who specializes in auditing blockchain technology independently. By identifying multiple security flaws in different protocols, he actively works towards enhancing the security of the blockchain ecosystem through thorough research and reviews. Feel free to reach out on Twitter @hashovweb3

About T-Swap

Link to the project can be found here.

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

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - e643a8d4c2c802490976b538dd009b351b1c8dda**

Scope

The following smart contracts were in scope of the audit:

- **`PoolFactory.sol`**
- **`TSwapPool.sol`**

The following number of issues were found, categorized by their severity:

- Critical 0 issues
- High: 2 issues
- Medium: 1 issues
- Low: 5 issues
- Informational: 2 issues

Findings Summary

ID	Title	Severity
[H-01]	Unused of `deadline` parameter inside `deposit()` function can lead to unexpected deposits	High
[H-02]	Lack of constant usage leads to an extra `0` inside fee percentage calculation	High
[M-01]	Wrong usage of `poolTokenAmount` param for `swapExactOutput()` inside `sellPoolTokens()`	Medium
[L-01]	Wrong function return type for `swapExactInput()`	Low
[L-02]	Wrong parameter placement for `LiquidityAdded` event in `_addLiquidityMintAndTransfer()`	Low
[L-03]	State changes after external contract call in `deposit()`	Low
[L-04]	Misusage of liquidity token `name` instead of `symbol` in `createPool()` function	Low
[L-05]	Constructor initialization of `Poolfactory.sol` missing zero address check	Low
[I-01]	Usage of magic numbers instead of constants inside `TSwapPool.sol`	Info
[I-02]	Unused `PoolFactory__PoolDoesNotExist` error inside `PoolFactory.sol`	Info

Detailed Findings

[H-01] Unused of `deadline` parameter inside `deposit()` function can lead to unexpected deposits

Severity

Impact: High, as it results in unexpected protocol deposits.

Likelihood: High, as any user of **`TSwap Protocol`** can deposit at any time.

Description

Function `deposit()` does not use `deadline` parameter, therefore anyone can deposit at anytime, so if user expects his deposit to fail it will probably pass, cause no check is made based on that deadline this will be causing unexpected behaviour for the user.

Recommendations

Add a check if deadline passed such as:

```
require(block.timestamp < deadline, "Deadline has passed, no deposits can be made!");
```

[H-02] Lack of constant usage leads to an extra `0` inside fee percentage calculation

Severity

Impact: High, as it results in really high fee percentage for the liquidity provider.

Likelihood: High, as any time `getInputAmountBasedOnOutput()` is called will be calculated this way.

Description

Function `getInputAmountBasedOnOutput()` inside of `TSwapPool.sol` has a fee calculation for the liquidity providers such as:

```
return((inputReserves * outputAmount) * 10000)/((outputReserves - outputAmount) * 997);
```

and as it can be seen there are some magic numbers that have to be extracted in constant, since the same numbers are used inside `getOutputAmountBasedOnInput()` for fee

calculation, if a constant variable was used an extra `0` to the 1000 wouldn't have been added.

Recommendations

Don't use magic numbers, use constant instead.

[M-01] Wrong usage of `poolTokenAmount` param for `swapExactOutput()` inside `sellPoolTokens()`

Severity

Impact: Medium, as it will mistake what has to be sent with what has to be received for the user.

Likelihood: High, as every time you call `swapExactOutput()` it will happen.

Description

Inside of `TSwapPool.sol` the client is trying to sell his pool tokens in exchange for WETH, the method that should be called is `swapExactInput()`, cause `swapExactOutput()` 3d parameter is `uint256 outputAmount`, which is wanted to be WETH, but we do pass `poolTokenAmount` which we want to be the input instead.

Recommendations

`swapExactOutput()` to be changed with `swapExactInput()`

[L-01] Wrong function return type for `swapExactInput()`

Severity

Impact: Low, user will be confused on what he will expect

Likelihood: High, every time the `swapExactInput()` is called user will have confusion.

Description

`swapExactInput()` function inside of `TSwapPool.sol` has wrong return type of `uint256 output` might cause confusion to client.

Recommendations

Remove the return type.

[L-02] Wrong parameter placement for `LiquidityAdded` event in `_addLiquidityMintAndTransfer()`

Severity

Impact: Low, will have consumer confused.

Likelihood: High, as every time event is emitted parameters will be swapped.

Description

Inside of `_addLiquidityMintAndTransfer()` of `TSwapPool.sol` it is seen that an event is getting emitted such as:

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

and the method arguments have `poolTokensToDeposit` and `wethToDeposit` with swapped places, should be the other way around.

Recommendations

Swap the parameter places.

[L-03] State changes after external contract call in `deposit()`

Severity

Impact: Low, as no reentrancy will be present.

Likelihood: High, as it will happen each time client goes through this code piece.

Description

Inside `TSwapPool.sol` we have have this piece of code:

```
_addLiquidityMintAndTransfer(wethToDeposit,maximumPoolTokensToDeposit,wethToDeposit);

liquidityTokensToMint = wethToDeposit;
```

No sign of reentrancy since *`ERC-20`* and especially "*Openzeppelin*"s implementation of *`SafeERC-20`* are safe to use, but it is a good practice to have state changes before external contracts calls, so `liquidityTokensToMint = wethToDeposit;` should be before `_addLiquidityMintAndTransfer();`

Recommendations

Swap line for: `liquidityTokensToMint = wethToDeposit;` and `_addLiquidityMintAndTransfer();`

[L-04] Misusage of liquidity token `name` instead of `symbol` in `createPool()` function

Severity

Impact: Low, as it will not have almost no impact to client except wrong naming of liquidity token.

Likelihood: High, as this will happen any time `createPool()` is called

Description

the `createPool()` inside `PoolFactory.sol` has this code piece:

```
string memory liquidityTokenName = string.concat("T-Swap ",
IERC20(tokenAddress).name());
string memory liquidityTokenSymbol = string.concat("ts",
IERC20(tokenAddress).name());
```

where the liquidity token metadata is being set and for the `liquidityTokenSymbol` we have concatenation of `ts + IERC20(tokenAddress).name()` instead of `IERC20(tokenAddress).symbol()`.

Recommendations

Change `IERC20(tokenAddress).name()` to `IERC20(tokenAddress).symbol()`

[L-05] Constructor initialization of `Poolfactory.sol` missing zero address check

Severity

Impact: High, as it will lead of client losing ownership of contract.

Likelihood: Low, cause the chance of contract initializer calling constructor with 0 address is really low.

Description

Constructor inside of `Poolfactory.sol` missing zero check for address, which means user can instantiate the contract without specifying address and will lose ownership of the contract.

Recommendations

```
constructor(address wethToken) {
    require(wethToken != address(0), "Address 0 not permitted!");
    i_wethToken = wethToken;
}
```

[I-01] Usage of magic numbers instead of constants inside `TSwapPool.sol`

Severity

Description

The `TSwapPool.sol` is having some magic numbers inside of `getOutputAmountBasedOnInput()` that is a best practice to be constants.

```
uint256 inputAmountMinusFee = inputAmount * 997;
uint256 denominator = (inputReserves * 1000) + inputAmountMinusFee;
```

Recommendations

Add constants like:

```
uint8 private constant FEE_PERCENTAGE = 997;
```

[I-02] Unused `PoolFactory__PoolDoesNotExist` error inside `PoolFactory.sol`

Severity

Informational

Description

Inside PoolFactory.sol there is an error declaration such as:

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

which is unused, has to removed.

Recommendations

Removed unused variables, safe gas.