TSWAP

SECURITY AUDIT REPORT::2024-01-18

SAFIRAETHER.COM
SAFEGUARDING ETHEREUM

Prepared by 808Nestor
NESTOR@SAFIRAETHER.COM

TABLE OF CONTENTS

INTRODUCTION

About The Protocol

Protocol Objective

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 like Uniswap.

The protocol starts as simply a PoolFactory contract. This contract is used to create new "pools" of tokens. It helps make sure every pool token uses the correct logic. But all the magic is in each TSwapPool contract.

You can think of each TSwapPool contract as it's own exchange between exactly 2 assets. Any ERC20 and the WETH token. These pools allow users to permissionlessly swap between an ERC20 that has a pool and WETH. Once enough pools are created, users can easily "hop" between supported ERC20s.

Solc Version

0.8.20

Chains

Ethereum

Tokens

Any ERC20 token

Audit Scope

Commit Hash

e643a8d4c2c802490976b538dd009b351b1c8dda

Files in Scope

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

Audit Roles

1. 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.
2. Users: Users who want to swap tokens.

Known Issues

None

Disclaimer

Methodology

Our audit methodology focused on ensuring the security, reliability, and adherence to best practices in the context of the Ethereum blockchain.

Business Logic Assessment

Our analysis commenced with a thorough understanding of the smart contract's business logic. The goal was to identify the core functionalities and interactions with external components, laying the foundation for subsequent evaluations.

Manual Code Review

A meticulous review of the Solidity source code was conducted, adhering to industry best practices and coding standards. The purpose was to identify potential vulnerabilities, ensuring the code is robust and maintainable.

Automated Analysis

Advanced automated analysis tools, including Aderyn, and Slither, were employed to identify common vulnerabilities related to security, gas efficiency, and code style. The results contributed to a comprehensive understanding of potential risks. Additionally, testing of invariants was conducted where appropriate via fuzzing and formal verification using tools such as Halmos and Certora.

Security Patterns Application

Security patterns and anti-patterns were applied to address common vulnerabilities, including reentrancy, overflow/underflow, and timestamp dependency. Access controls and permissions were implemented judiciously to enhance overall security.

External Calls Evaluation

External calls to other contracts or external systems were scrutinized to mitigate the risk of reentrancy attacks. The assessment ensured that all calls were secure and aligned with the integrity of the contract.

Input Validation

Rigorous validation of user inputs was conducted to ensure the smart contract gracefully handles unexpected inputs, guarding against vulnerabilities such as integer overflow and underflow.

Risk Classifications

Risk is classified based on two factors: likelihood and impact.

Likelihood

Likelihood refers to the probability of a specific event or vulnerability being exploited. Here's an elaboration on the three levels of likelihood:

High Likelihood:

In situations of high likelihood, the conditions for exploitation are readily accessible or the attack vector is easily achievable. The vulnerability is considered highly exploitable, and the likelihood of occurrence is relatively high.

Example Scenario: A scenario where a hacker can directly call a function to cause a significant impact on the smart contract's behavior. This could involve straightforward and easily executable steps that do not require elaborate conditions.

Medium Likelihood:

In cases of medium likelihood, specific conditions or a more constrained set of circumstances are required for the vulnerability to be exploited. While not as easily achievable as high likelihood scenarios, the conditions for exploitation are still reasonably attainable, making the event moderately likely to occur.

Example Scenario: An example could be a vulnerability that depends on the use of a particular type of token within the platform. While not universally applicable, the conditions for exploitation are plausible and could occur under specific circumstances.

Low Likelihood:

Vulnerabilities with low likelihood are associated with rare situations that are unlikely to happen in typical scenarios. Although technically feasible, the conditions required for exploitation are infrequent or involve a combination of events that are unlikely to align.

Example Scenario: Consider a vulnerability that relies on a unique sequence of events (A, B, C) taking place at a specific time. While technically possible, the occurrence of such a sequence is rare and unlikely, making the exploitation of the vulnerability less probable.

NOTE:

Computationally Unfeasible: Events that are 'computationally unfeasible' are practically impossible due to their extreme rarity or the astronomical computational effort required for exploitation. These are not considered viable attack paths in practice.

Assessing likelihood involves a degree of subjectivity and requires a comprehensive understanding of the smart contract's architecture, the blockchain environment, and potential threat vectors. Regular reviews and updates to likelihood assessments should be conducted to adapt to changes in the threat landscape and the evolving nature of decentralized applications.

IMPACT

Impact is a crucial aspect of vulnerability assessment, representing the potential harm or consequences resulting from a vulnerability. Here are the three levels of impact:

HIGH IMPACT:

High impact vulnerabilities pose a significant threat to the protocol, where funds are directly or nearly directly at risk. The consequences involve a severe disruption of protocol functionality or availability, with potential financial losses for users.

Examples:

- Direct exposure of funds to unauthorized access.
- Severe disruption in the protocol's core functionality, leading to financial losses.

MEDIUM IMPACT:

Medium impact vulnerabilities introduce some level of risk to funds, albeit indirectly. While not as severe as high impact scenarios, these vulnerabilities can still result in disruptions to the protocol's functionality or availability, impacting the user experience and potentially leading to financial consequences.

Examples:

- Indirect risk to funds due to a vulnerability that affects the integrity of transactions or user accounts.
- Moderate disruption in protocol functionality, affecting user interactions and potentially leading to financial consequences.

Low Impact:

Low impact vulnerabilities do not directly put funds at risk. However, there might be issues related to the correctness of functions, inappropriate handling of states, or other non-financial implications. The primary concern is related to the correctness and reliability of the protocol rather than immediate financial losses.

Examples:

- Correctness issues in specific functions that do not pose a direct threat to funds but may impact the overall reliability of the protocol.
- State handling concerns that do not immediately impact financial transactions but may affect the protocol's overall performance.

Assessing impact is crucial for prioritizing the resolution of vulnerabilities and allocating resources effectively. It helps in understanding the potential consequences of each vulnerability and guides the development team in addressing the most critical issues first, ensuring the overall security and stability of the protocol. Regular impact assessments should be conducted to adapt to changes in the protocol's features and user interactions.

Severity Ratings

Severity ratings in the context of smart contract security assessments are typically derived from the combination of likelihood and impact assessments. Here are the severity ratings based on the provided likelihood and impact descriptions:

	High	MEDIUM	HIGH	CRITICAL
Impact	**Medium**	LOW	MEDIUM	HIGH
	Low	LOW	LOW	MEDIUM
		Low	**Medium**	**High**
			Likelihood	

Critical Severity:

- High Likelihood + High Impact

Vulnerabilities with a critical severity rating represent a high likelihood of exploitation coupled with significant consequences. In these scenarios, funds are directly at risk, and there's a severe disruption of protocol functionality or availability, posing a substantial threat to users and the overall integrity of the protocol.

High Severity:

- Medium Likelihood + High Impact
- High Likelihood + Medium Impact

High severity vulnerabilities indicate a substantial risk to the protocol. While the likelihood may be high or medium, the impact is never low, with the potential for direct financial losses and severe disruption of protocol functionality. These vulnerabilities demand immediate attention and remediation efforts.

Medium Severity:

- Low Likelihood + High Impact
- Medium Likelihood + Medium Impact
- High Likelihood + Low Impact

Medium severity vulnerabilities pose a moderate risk to the protocol. While the likelihood of exploitation and the impact may vary, the potential for indirect financial risks and disruptions to the protocol's functionality remains constant. Timely remediation is recommended to maintain the overall security and stability of the system.

Low Severity:

- Low Likelihood + Low Impact
- Low Likelihood + Medium Impact
- Medium Likelihood + Low Impact

Low severity vulnerabilities represent a lower risk to the protocol. The likelihood of exploitation is low or medium, and the impact may be low or medium, primarily related to correctness issues or non-financial implications. While these vulnerabilities are not immediate threats, they should still be addressed in a timely manner to enhance the robustness of the protocol.

Informational Severity:

- Likelihood: Not Applicable
- Impact: Not Applicable

Informational severity is used for findings that provide valuable information but do not pose an immediate risk to the protocol's security or functionality. These findings may include suggestions for improvement or best practices that could enhance the overall security posture.

FINDINGS

EXECUTIVE SUMMARY

OVERVIEW

In the TSwap smart contract audit, several critical findings have been identified.

First, in the TSwapPool's deposit function, a crucial deadline check is missing, allowing transactions to complete even after the specified deadline. This could pose a significant problem if users add money to the pool at unexpected times, especially when market conditions are unfavorable for making deposits.

Additionally, there's an issue with fee calculation in the TSwapPool, where the getInputAmountBasedOnOutput function multiplies the fee amount by 10,000 instead of the correct value, 1,000, resulting in the protocol taking too many tokens from users and causing them to lose more fees than intended.

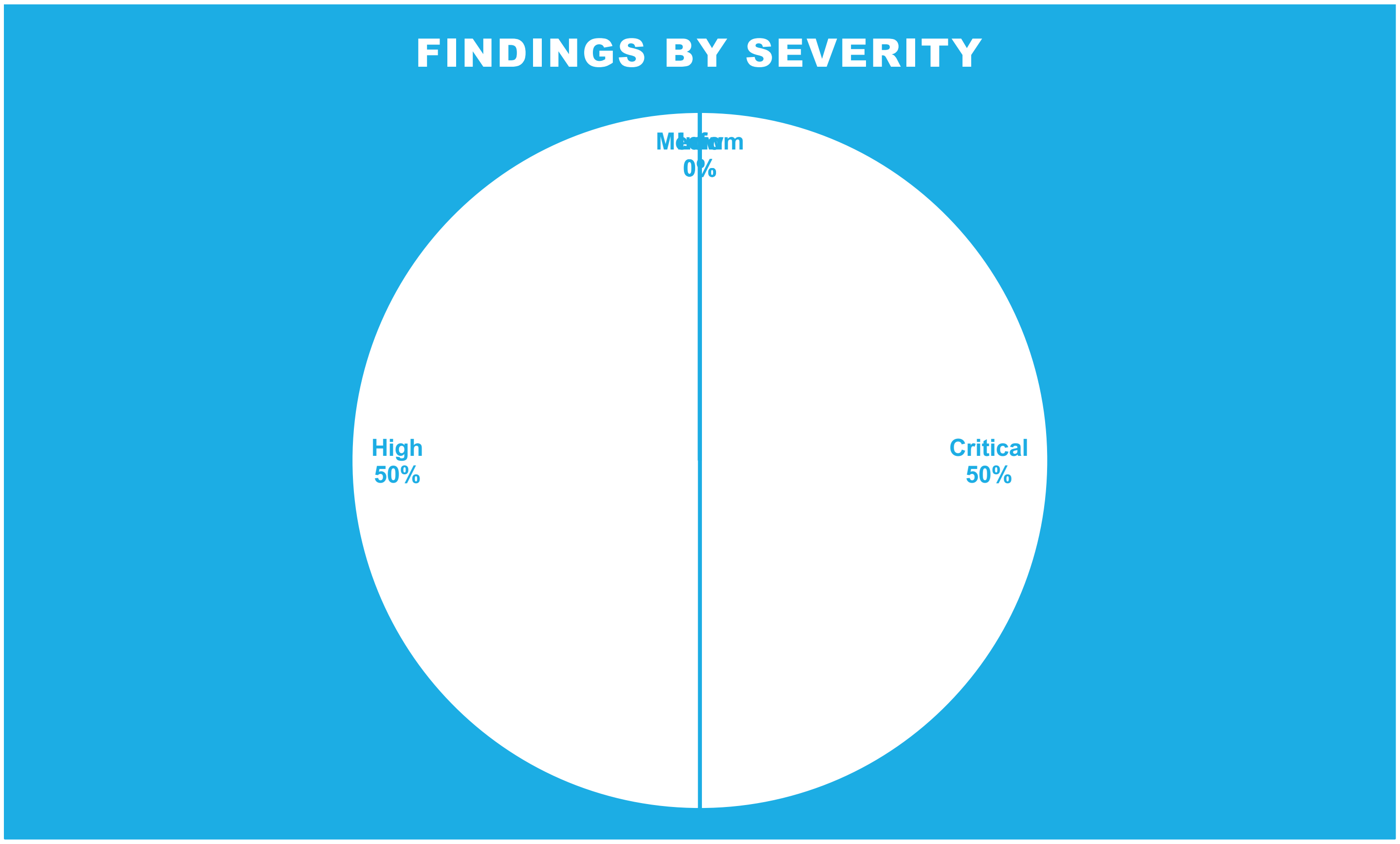
Furthermore, the swapExactOutput function in TSwapPool lacks slippage protection, potentially leading users to receive significantly fewer tokens than expected if market conditions change during the transaction.

Finally, the sellPoolTokens function suffers from a mismatch between input and output tokens, causing users to receive an incorrect number of tokens when selling pool tokens. These critical issues need urgent attention to ensure the security and functionality of the TSwap protocol.

Findings By Severity Tally Table

Severity	Tally
Critical	2
High	2
Medium	0
Low	0
Info	0
TOTAL	4

Findings By Severity Tally Chart



Critical Severity Findings

C-01: Missing deadline parameter in TSwapPool::deposit results in allowing transactions to complete after deadline.

File: TSwapPool

Element: deposit()

Likelihood: High

Financial Impact: High

Severity: Critical

Details

The TSwapPool's deposit feature has a flaw. It doesn't check the deadline, which is like a time limit for the transaction to finish. The issue is that even if a deadline exists, the function is unaware of it. This means that when people are adding money to the pool, it might happen at times they don't expect. This could be a problem if the market conditions are not favorable for making deposits.

Impact

The impact of this issue is that people could lose money if they are not careful. If the market conditions are not favorable, they could lose money.

Proof Of Concept

```
The deadline parameter is unused.
```

Tools Used

Manual

Recommendations

Add a deadline parameter to the deposit function.

C-02: Incorrect fee calculation in TSwapPool::getInputAmountBasedOnOutput results in lost fees.

File: TSwapPool

Element: getInputAmountBasedOnOutput()

Likelihood: High

Financial Impact: High

Severity: Critical

Details

There's an issue with the way fees are calculated in the TSwapPool. The function called getInputAmountBasedOnOutput is supposed to figure out how many tokens a user needs to deposit based on the amount of output tokens they want. However, there's a mistake in the calculation. When it comes to figuring out the fee, it multiplies the amount by 10,000 instead of the correct value, which is 1,000. This mistake means that the protocol is taking too many tokens from users, and as a result, users are losing more fees than they should.

Impact

The impact of this issue is that users are losing more fees than they should.

Proof Of Concept

```
return ((inputReserves * outputAmount) * 10_000) / ((outputReserves - outputAmount) * 997);
```

Tools Used

Manual

Recommendations

Change the 10,000 to 1,000 in the getInputAmountBasedOnOutput function.

High Severity Findings

H-01: Lack of slippage protection in TSwapPool::swapExactOutput results in lost funds.

File: TSwapPool

Element: swapExactOutput()

Likelihood: Medium

Financial Impact: High

Severity: High

Details

There's a problem with the swap function in TSwapPool. Specifically, the swapExactOutput function doesn't have any

protection against slippage. In simpler terms, slippage protection helps users from getting significantly fewer

tokens than expected. Here's what happens: When users execute swapExactOutput, the function doesn't set a maximum amount for the input tokens. This is important because if market conditions change before the transaction is complete, users might end up

with a much worse deal than they anticipated.

Impact

The impact of this issue is that users could lose money. If

market conditions change before the transaction is complete, users could end up with a much worse deal than they anticipated.

Proof Of Concept

1. The price of 1 WETH right now is 1,000 USDC
2. User inputs a `swapExactOutput` looking for 1 WETH
    i. inputToken = USDC
    ii. outputToken = WETH
    iii. outputAmount = 1
    iv. 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

Tools Used

Manual

Recommendations

Add slippage protection to the swapExactOutput function by specifying a maximum input amount.

```
   function swapExactOutput(
      IERC20 inputToken,
+      uint256 maxInputAmount,
.
.
.
      inputAmount = getInputAmountBasedOnOutput(outputAmount, inputReserves,
outputReserves);
+      if(inputAmount > maxInputAmount){
+          revert();
+      }
      _swap(inputToken, inputAmount, outputToken, outputAmount);
```

H-02: Mismatch between input and output in TSwapPool::sellPoolTokens results in lost funds.

File: TSwapPool

Element: sellPoolTokens()

Likelihood: Medium

Financial Impact: High

Severity: High

Details

In the TSwapPool contract, the sellPoolTokens function is designed to simplify the process for users looking to sell pool tokens and receive WETH in return.

However, there is an issue causing a mismatch between the input and output tokens, leading users to receive an incorrect amount. The problem arises from using the swapExactOutput function instead of the appropriate swapExactInput function.

In this context, users specify the number of pool tokens they want to sell (input), but the function incorrectly calculates the swapped amount as if users were specifying the exact amount of output tokens.

Impact

This discrepancy results in users swapping an incorrect amount of tokens, significantly disrupting the functionality of the protocol.

Proof Of Concept

```
return swapExactOutput(i_poolToken, i_wethToken, poolTokenAmount, uint64(block.timestamp));
```

Tools Used

Manual

Recommendations

Modify the sellPoolTokens function to use the swapExactInput function, ensuring accurate token swaps for users. In other words, use the swapExactInput function instead of swapExactOutput.

```
    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));
    }
```

Medium Severity Findings

N/A

Low Severity Findings

N/A

Informational Level Findings

N/A

CONCLUSION

In conclusion, the smart contract audit for TSwap has revealed several critical findings that demand immediate attention.

The TSwapPool's deposit function lacks a crucial deadline check, allowing transactions to complete after the specified deadline, potentially causing issues when users add funds to the pool.

Additionally, the TSwapPool's fee calculation in the getInputAmountBasedOnOutput function is flawed, multiplying the fee amount by 10,000 instead of the correct value, leading to users losing more fees than intended.

Furthermore, the swapExactOutput function in TSwapPool lacks slippage protection, potentially resulting in users receiving significantly fewer tokens than expected if market conditions change during the transaction.

Lastly, the sellPoolTokens function suffers from a mismatch between input and output tokens, causing users to receive an incorrect number of tokens when selling pool tokens.

These critical issues pose a substantial risk to the functionality and security of the TSwap protocol and require immediate corrective measures to ensure the integrity of the platform.

APPENDICES

N/A