QuillAudits

# Audit Report
# January, 2022

For

alium finance

# Contents

## Overview

AliumSwap is a fork of PancakeSwap, but implements a fee mechanism with swaps.

## Scope of Audit

The scope of this audit was to analyse **AliumSwap smart contract**'s codebase for quality, security, and correctness.

**AliumSwap Contracts**: AliumSideSwapWithPancakeRouter
**Branch: feature/multi-factory**
**Commit: 6aff4c5ad4695b77358eae0627c98d72621256bc**
**Fixed In:** AliumSideSwapWithPancakeRouter
**Branch: master**
**Commit : be309b2c919faf32973a822a17760cbcf5af9ad2**

## Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC20 transfer() does not return
- boolean
- ERC20 approve() race
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

## Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20/BEP20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

**Structural Analysis**
In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

**Static Analysis**
Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

**Code Review / Manual Analysis**
Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

**Gas Consumption**
In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

**Tools and Platforms used for Audit**
Mythril, Slither, SmartCheck, Surya, Solhint.

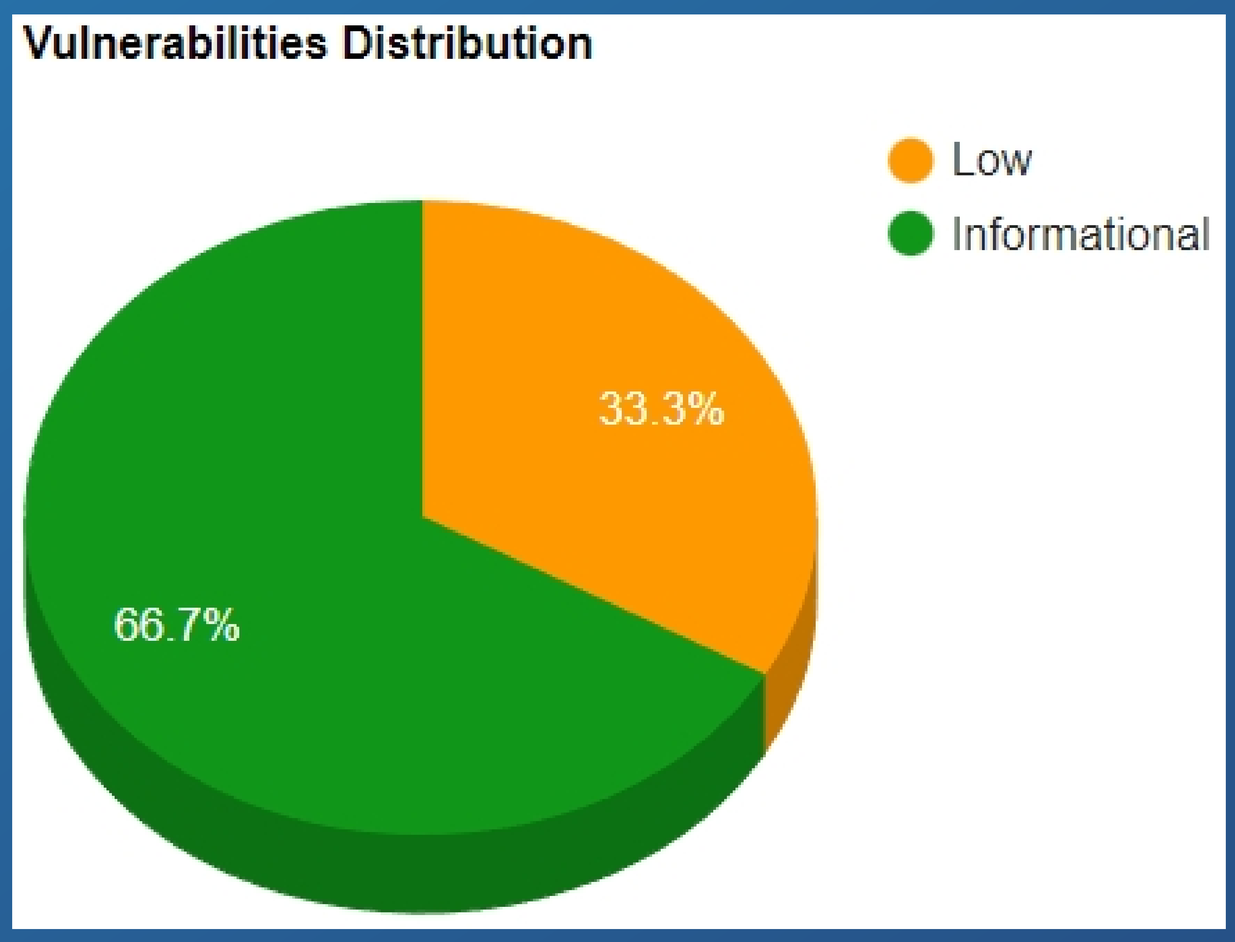
# Issue Categories

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

| Risk-level | Description |
|---|---|
| High | A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment. |
| Medium | The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed. |
| Low | Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future. |
| Informational | These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact. |

## Number of issues per severity

| Type | High | Medium | Low | Informational |
|---|---|---|---|---|
| Open | 0 | 0 | 0 | 0 |
| Acknowledged | 0 | 0 | 1 | 2 |
| Closed | 0 | 0 | 1 | 1 |




## Functional Testing Results

```
should deploy
  ✓ should deploy
  ✓ should add liquidity of tokens (203ms)
  0 fees, should not deduct any amount
    ✓ swapExactTokensForTokens (53ms)
    ✓ swapTokensForExactTokens (56ms)
    ✓ swapExactETHForTokens (58ms)
    ✓ swapTokensForExactETH (45ms)
    ✓ swapExactTokensForETH (45ms)
    ✓ swapETHForExactTokens (57ms)
  1000 fees, should deduct 10% fees from the amount
    ✓ swapExactTokensForTokens (42ms)
    ✓ swapTokensForExactTokens (55ms)
    ✓ swapExactETHForTokens (38ms)
    ✓ swapTokensForExactETH (48ms)
    ✓ swapExactTokensForETH (54ms)
    ✓ swapETHForExactTokens (42ms)


  14 passing (3s)
```

# Issues Found – Code Review / Manual Testing

## High severity issues

No issues were found.

## Medium severity issues

No issues were found.

## Low severity issues

1. **Missing two step process to change privileged role Owner**

   When privileged roles are being changed, it is recommended to follow a two-step approach: 1) The current privileged role proposes a new address for the change 2) The newly proposed address then claims the privileged role in a separate transaction. This two-step change allows accidental proposals to be corrected instead of leaving the system operationally with no/malicious privileged role. (Ref: Security Pitfalls: 162)

   However, token contract uses **Ownable** contract to define a privileged role owner, which uses a one step process to transfer/renounce ownership, as a consequence the owner may accidently lose control over the operational logic of the contract.

   **Recommendation**
   Consider switching to a two-step process for transferring ownership and an optional time-margin for renouncing ownership.

   **Status:** Acknowledged

### 2. Missing Zero Address Validation

The contract lacks zero address checks for _factory and _WETH,

```
524         constructor(address _factory, address _WETH) public {
525             factory = _factory;
526             WETH = _WETH;
527         }
528
```

As a consequence of which the contract is prone to incorrect initialization.

**Recommendation**
Consider adding the required checks.

**Status:** Fixed

## Informational issues

### 3. Misleading operational logic

As the contract implements a fee mechanism for token swaps, the implementation affects the intended operational logic of some functions.

For ref., **swapTokensForExactTokens**
The function’s intended logic is to give the user an Exact number of desired tokens after the swap. However, implemented fee mechanism deducts some fees from the desired amount, and the calculation happens on the resultant value and not on the desired value.

```
750     function swapTokensForExactTokens(
751         uint amountOut,
752         uint amountInMax,
753         address[] calldata path,
754         address to,
755         uint deadline
756     ) external virtual override ensure(deadline) returns (uint[] memory amounts) {
757         uint256 toDev = fee * amountOut / DECIMAL;
758         if (toDev > 0 && feeTo != address(0)) {
759             TransferHelper.safeTransferFrom(
760                 path[0],
761                 msg.sender,
762                 feeTo,
763                 toDev
764             );
765         }
766         amounts = PancakeLibrary.getAmountsIn(factory, amountOut - toDev, path);
767         require(amounts[0] <= amountInMax, 'PancakeRouter: EXCESSIVE_INPUT_AMOUNT');
768         TransferHelper.safeTransferFrom(
769             path[0], msg.sender, PancakeLibrary.pairFor(factory, path[0], path[1]), amounts[0]
770         );
771         _swap(amounts, path, to);
772     }
```

For. eg, let’s assume fees set is 10%, and a user wants 500 output tokens i.e **amountOut**. Instead of calculating how many tokens will be required for 500 output tokens, the function after deducting fees, will calculate the required number of input tokens over (500 - 10% of 500) i.e 450 output tokens, thus affecting the intended meaning/logic of the function, because a user would not be getting the Exact desired tokens as the function name suggests.

Similar affect can be observed in the function swapTokensForExactETH

**Recommendation**
Consider reviewing and verifying the operational and business logic

**Status:** Acknowledged

**Developer Comments:** “All info about addition fee will be shown on the front APP”

### 4. Getter functions not accounting fee factor

The contract implements a fee mechanism for token swaps. Also, there exists multiple getter functions for the users in order to calculate input/output amount of tokens prior to the swap. However, these functions don’t account for the fee factor, which may produce incorrect results until and unless the user accounts for this fee factor manually.

These functions are:

```
946     function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut)
947     public
948     pure
949     virtual
950     override
951     returns (uint amountOut)
952     {
953         return PancakeLibrary.getAmountOut(amountIn, reserveIn, reserveOut);
954     }
955
956     function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut)
957     public
958     pure
959     virtual
960     override
961     returns (uint amountIn)
962     {
963         return PancakeLibrary.getAmountIn(amountOut, reserveIn, reserveOut);
964     }
965
966     function getAmountsOut(uint amountIn, address[] memory path)
967     public
968     view
969     virtual
970     override
971     returns (uint[] memory amounts)
972     {
973         return PancakeLibrary.getAmountsOut(factory, amountIn, path);
974     }
975
976     function getAmountsIn(uint amountOut, address[] memory path)
977     public
978     view
979     virtual
980     override
981     returns (uint[] memory amounts)
982     {
983         return PancakeLibrary.getAmountsIn(factory, amountOut, path);
984     }
```

**Recommendation**
Consider reviewing and verifying the operational and business logic

**Status:** Acknowledged

**Developer Comments:** "The fee will be programmatically handled on the front APP"

5. **Multiple pragma directives have been used.**

Recommendation: Contracts should be deployed using the same compiler version/flags with which they have been tested. Locking the pragma (for e.g. by not using ^ in pragma solidity 0.5.10) ensures that contracts do not accidentally get deployed using an older compiler version with unfixed bugs. Ref: Security Pitfall 2

**Status:** Fixed

# Closing Summary

Some issues of Low severity were found. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

# Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the **AliumSwap** contract. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the **AliumSwap** team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

# Audit Report
# January, 2022

For

alium
finance

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com