QuillAudits

Audit Report
September, 2024

For

MATCHAIN

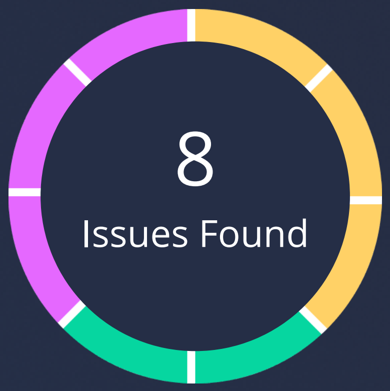
Table of Content

Table of Content

Executive Summary

Project Name	Matchain
Project URL	https://www.matchain.io/
Overview	Matchain is a fork of Uniswap V2, It maintains the core structure and functionality of Uniswap V2, including the automated market maker (AMM) system, liquidity provision, token swaps, and flash swap capabilities. The main contracts are MSwapFactory for pair creation and management, MSwapPair for handling liquidity pairs and swaps, and MSwapERC20 for the implementation of liquidity tokens.
Audit Scope	The Scope of the Audit was to check, security of MatchainCodebase for vulnerabilities and code quality.
Contracts In-Scope	https://github.com/MatchSwap/MSwap-contract/tree/main **Contracts:** Factory.sol Router.sol WBNB.sol
Commit Hash	96a1af240ba5a690d9875ff5f11431bdc4942971
Language	Solidity
Blockchain	Binance
Method	Manual Analysis, Functional Testing, Automated Testing
First Review	26th August 2024 - 2nd September 2024
Updated Code Received	NA
Second Review	NA
Fixed In	NA

Number of Security Issues per Severity



🟥 High
🟨 Medium
🟩 Low
🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	3	2	3
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	0	0	0

Checked Vulnerabilities

✓ Access Management	✓ Compiler version not fixed
✓ Arbitrary write to storage	✓ Address hardcoded
✓ Centralization of control	✓ Divide before multiply
✓ Ether theft	✓ Integer overflow/underflow
✓ Improper or missing events	✓ ERC's conformance
✓ Logical issues and flaws	✓ Dangerous strict equalities
✓ Arithmetic Correctness	✓ Tautology or contradiction
✓ Race conditions/front running	✓ Return values of low-level calls
✓ SWC Registry	✓ Missing Zero Address Validation
✓ Re-entrancy	✓ Private modifier
✓ Timestamp Dependence	✓ Revert/require functions
✓ Gas Limit and Loops	✓ Multiple Sends
✓ Exception Disorder	✓ Using suicide
✓ Gasless Send	✓ Using delegatecall
✓ Use of tx.origin	✓ Upgradeable safety
✓ Malicious libraries	✓ Using throw

Checked Vulnerabilities

- ✓ Using inline assembly
- ✓ Unsafe type inference
- ✓ Style guide violation
- ✓ Implicit visibility level

Techniques and Methods

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

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC 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 analyzed 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

A 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 analyzed, 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 behavior 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

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity statistic analysis.

Types of Severity

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.

High Severity Issues

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 Severity Issues

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 Severity Issues

Low-level severity issues can cause minor impact and 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 four 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.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Medium Severity Issues

1. A malicious user can grief and DOS a user while he is trying to remove Liquidity

Path

Router.sol

Function

removeLiquidityWithPermit(),removeLiquidityETHWithPermit()

Description

The removeLiquidityWithPermit() and removeLiquidityETHWithPermit() use permit function to give approve to router contract for the LP tokens and removal of liquidity in the same transaction rather than approving the LP token manually from contract address.

- User signs their LP tokens and provides the signature(v,r,s) to contract as part of approving token using permit.
- When a user calls removeLiquidityWithPermit or removeLiquidityETHWithPermit function the malicious user see's the transaction in mempool and take the signature of user and calls the permit function himself before the user's transaction gets executed.
- Since the signature provided by malicious user is correct the transaction get's successful and increases the nonce.
- However, when user's transaction gets executed it would fail due to increase of nonce.

Recommendation

try IERC20Permit(pair).permit(msg.sender, address(this), amount, deadline, v, r, s) {} catch {}

Status

Acknowledged

2. Using payable.transfer might be problematic

Path

WBNB.sol

Function

withdraw()

Description

The usage of transfer so send msg.value is not recommended due to it’s strict dependency upon gas i.e. 2300 gas.
If gas costs are subject to change, then smart contracts can’t depend on any particular gas costs.

Moreover, the function might fail mid-execution is it requires more than 2300 gas.

Recommendation

Switch to call() instead

Status

Acknowledged

3. Different solidity versions used may cause vulnerability

Description

The contract specifies multiple conflicting Solidity versions in its pragma statements. This is contradictory and will cause compilation errors, as a contract can only be compiled with one specific version of Solidity. Using different Solidity versions may introduce vulnerabilities.

Recommendation

Use the same solidity version

Status

Acknowledged

Low Severity Issues

4. Use an outdated version of the solidity

Description

Versions 0.5.16 and especially 0.4.18 are quite old and lack important security features and optimizations present in newer versions. Using outdated compiler versions can lead to security vulnerabilities.

Recommendation

Use the latest version of solidity

Status

Acknowledged

5. Low test coverage

Description

Unit tests are used to ensure that the code functions as expected by passing in varying user input and setting up various parameters to test the boundaries of the protocol. There are no unit test cases associated with the codebase provided, hereby increasing the probability of bugs being present and reducing quality assurance.

Recommendation

Include unit tests that have > 95% code coverage, including all possible paths for code execution.

Status

Acknowledged

Informational Issues

6. Inconsistent version specifiers

Description

You're using both strict (=) and caret (^) version specifiers. The caret allows any version from 0.4.18 up to, but not including, 0.5.0, which could lead to unexpected behavior if the contract is compiled with different minor versions.

Recommendation

Generally, using ^ is preferred for development.

Status

Acknowledged

6. Inconsistent version specifiers

Description

- **Readability and Maintainability:** Combining all interfaces and libraries into a single contract can make the contract difficult to read and maintain. It's usually better to separate concerns by having distinct contracts for different functionalities.
- **Complexity:** A single contract handling all aspects of a protocol can become overly complex, making it harder to debug and test individual components.
- **Gas Costs:** Large contracts can be more expensive to deploy and interact with due to higher gas costs.

Recommendation

Modular Design: Break down the protocol into smaller, more manageable contracts. For example:

- Interfaces: Define your interfaces in separate files.
- **Libraries:** Place library code in its own contract.
- **Core Logic:** Keep the main protocol logic in a separate contract.

Status

Acknowledged

8.Initial liquidity provider is not able to withdraw full amount

Path

Factory.sol

Function

mint()

Description

To prevent the “first deposit attack” in the liquidity provision mechanism, the protocol currently transfers a fixed amount of MINIMUM_LIQUIDITY to the zero address. As a result, the initial liquidity provider (LP) receives K - 1000 wei of assets, rather than the full amount K that they deposited

Recommendation

Use virtual shares

Status

Acknowledged

Functional Tests Cases

Some of the tests performed are mentioned below:

WBNB.sol

- ✓ deposit function works as per the expectation.
- ✓ approve function overrides previous approval that is as per expectation.
- ✓ transfer and transferfrom function transfers funds as per exception.
- ✓ Any user is not able to transfer more than his balance.
- ✓ Any user is not able to Withdraw more than his balance.
- ✓ fallback function works properly

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of Matchain. We performed our audit according to the procedure described above.

Some issues of low,medium and informational severity were found. Some suggestions, gas optimizations and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Matchain. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Matchain. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of Matchain to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
September, 2024

For

MATCHAIN

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com