QuillAudits

# Audit Report
# April, 2022

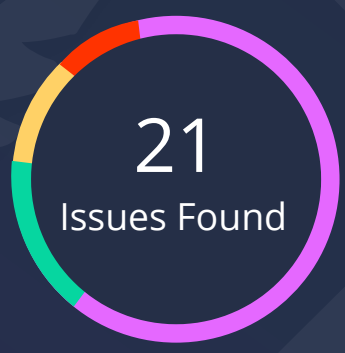
For

Neorder

# Table of Content

# Executive Summary

| | |
|---|---|
| **Project Name** | NeorderDao |
| **Overview** | NeorderDAO is such a decentralized autonomous organization, with the mission of establishing a new order for all digital immigrants in the era of Web 3.0 in socializing, communicating and advertising, returning the value achievement of peer- to-peer and node communication that should belong to all Internet users, allowing users to enjoy the traffic dividends created by themselves, and owning the brand new world transformed and built by the hands of all creative people. NeorderDAO will issue the identity and governance Token, N3DR, and capture the value created by the mass communication of this social experiment. |
| **Timeline** | 05 April, 2022 to 03 May, 2022 |
| **Method** | Manual Review, Functional Testing, Automated Testing etc. |
| **Scope of Audit** | The scope of this audit was to analyse NeoorderDao codebase for quality, security, and correctness. |
| **Source Code** | *https://github.com/neorder-io/contracts/tree/master/contracts/N3DR* |
| **Commit Hash** | 9cb33d1f06528ace02fd5c71ab994ab41c81455e |
| **Fixed In** | 246bf6ca5055f50d46d5f97d2ee45796c7c32f52 |




High
Medium
Low
Informational

| | High | Medium | Low | Informational |
|---|---|---|---|---|
| Open Issues | 0 | 0 | 0 | 0 |
| Acknowledged Issues | 0 | 1 | 7 | 9 |
| Partially Resolved Issues | 0 | 0 | 0 | 0 |
| Resolved Issues | 1 | 3 | 0 | 0 |

## Types of Severities

### 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.

## 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.

# Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- 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 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

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

# Manual Testing

## A. Contract - Farm.sol

## High Severity Issues

No Issues Found

## Medium Severity Issues

### A.1 transfer() is being utilized

**Description**

Low-level transfer() function has been found to be used in the contract at line 338:

_to.transfer(_amount);

Due to the fact that .transfer() and .send() forward exactly 2,300 gas to the recipient. This hardcoded gas stipend aimed to prevent reentrancy vulnerabilities, but this only makes sense under the assumption that gas costs are constant. Recently EIP 1884 was included in the Istanbul hard fork. One of the changes included in EIP 1884 is an increase to the gas cost of the SLOAD operation, causing a contract's fallback function to cost more than 2300 gas.

```
// bad
contract Vulnerable {
    function withdraw(uint256 amount) external {
        // This forwards 2300 gas, which may not be enough if the recipient
        // is a contract and gas costs change.
        msg.sender.transfer(amount);
    }
}

// good
contract Fixed {
    function withdraw(uint256 amount) external {
        // This forwards all available gas. Be sure to check the return value!
        (bool success, ) = msg.sender.call.value(amount)("");
        require(success, "Transfer failed.");
    }
}
```

**Remediation**

The auditee needs to ensure that the _to is not a contract .On the other hand, it's recommended to stop using .transfer() and .send() and instead use .call().

**Status**

**Fixed**

## Low Severity Issues

### A.2 Lack of event emissions

**Description**

The missing event makes it difficult to track off-chain liquidity fee changes. An event should be emitted for significant transactions calling the following functions:
- setPaused()
- setOperator()
- setTokenAddress()
- setPreDepositPid()
- addReward()

**Recommendation**

We recommend emitting an event to log the update of the above variables for those above-mentioned functions.

**Status**

**Acknowledged**

### A.3 Lack of Zero address validation

**Description**

To favor explicitness, consider adding a check for all functions that are taking address parameters in the entire codebase. Although most of the functions throughout the codebase properly validate function inputs, there are some instances of functions that do not. One example is:
- setOperator()
- setTokenAddress()
- rescue()

**Recommendation**

Consider implementing require statements where appropriate to validate all user-controlled input, including governance functions, to avoid the potential for erroneous values to result in unexpected behaviors or wasted gas.

**Status**

**Acknowledged**

## Informational Issues

### A.4 Public function that could be declared external

**Description**

The following public functions that are never called by the contract should be declared external to save gas:

- setPaused()
- setOperator()
- setTokenAddress()
- addPool()
- setPool()
- setPreDepositPid()
- addReward()
- disperseMainReward()
- disperseLpReward()
- multiPreDeposit()
- rescue()
- deposit()
- withdraw()

**Recommendation**

Use the external attribute for functions never called from the contract.

**Status**

**Acknowledged**

### A.5 Floating Pragma

**Description**

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Using floating pragma does not ensure that the contracts will be deployed with the same version. It is possible that the most recent compiler version get selected while deploying contract which has higher chances of having bugs in it.

**Remediation**

Lock the pragma version for the compiler version that is chosen.

**Status**

**Acknowledged**

## B. Contract - Lottery.sol

# High Severity Issues

No Issues Found

# Medium Severity Issues

### B.1 Possible Backrunning

**Description**

onTransfer() adds active orders in activeOrders mapping and removes the previous first elements if [#L170] _activeOrders.length > rewardRates.length condition becomes true , This can be exploited by an attacker to enter new orders at the last moment before [#L365]canDraw() becomes true and [#L258]draw() gets called.

**Description**

- The length of rewardRates is 10 and There are 10 active orders added in activeOrders when a any user address transfers tokens to other address.
- Assume that there's very less amount of time remaining for [#L365] canDraw() to become true so that lottery rewards can be assigned by [#L258]draw() successfully.
- Attacker now transfer some amount of N3DR tokens 10 times and N3DR:[#L520]_transfer() calls [#L130] onTransfer present in Lottery contract and adds 10 more orders at the last moment before [#L365] canDraw() becomes true
- This will remove previous 10 orders present in activeOrders because of this condition [#L170] _activeOrders.length > rewardRates.length and now there would be new 10 orders added by attacker.
- Once [#L258]draw() executes rewards get added for attacker addresses.Which then attacker can remove using [#L333]takeReward().
- In this way attacker can backrun all the active orders and can increase chances for getting selected for lottery before drawing the lottery as there would be only attacker address present in activeOrders mapping.

```
169         // remove first element from _activeOrders
170         if (_activeOrders.length > rewardRates.length) {
171             for (uint256 i = 0; i < _activeOrders.length - 1; i++) {
172                 _activeOrders[i] = _activeOrders[i + 1];
173             }
174             _activeOrders.pop();
175         }
176     }
177
```

**Recommendation**

Consider reviewing the code logic

**Status**

**Acknowledged**

**Comment from NeoorderDao Team:** “In case really if there are several times are the same attacker wins, naturally will cause other attackers to compete, so we believe its all right.”

## Low Severity Issues

### B.2 Lack of event emissions

**Description**

The missing event makes it difficult to track off-chain liquidity fee changes. An event should be emitted for significant transactions calling the following functions:
- setPaused()
- setOperator()
- setConfig()
- setRewardRates()

**Recommendation**

We recommend emitting an event to log the update of the above variables for those above-mentioned functions.

**Status**

**Acknowledged**

## Informational Issues

### B.3 Public function that could be declared external

**Description**

The following public functions that are never called by the contract should be declared external to save gas:

- setPaused()
- setOperator()
- setConfig()
- setRewardRates()
- rescue()
- takeReward()

**Recommendation**

Use the external attribute for functions never called from the contract.

**Status**

**Acknowledged**

### B.4 Floating Pragma

**Description**

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Using floating pragma does not ensure that the contracts will be deployed with the same version. It is possible that the most recent compiler version get selected while deploying contract which has higher chances of having bugs in it.

**Remediation**

Lock the pragma version for the compiler version that is chosen.

**Status**

**Acknowledged**

## C. Contract - N3DR.sol

# High Severity Issues

### C.1 Owner can burn user's token without their consent

**Description**

Owner can burn anyone's token.

```
417
418     function burn(address account, uint256 amount) public onlyOwner {
419         _burn(account, amount);
420     }
421
```

**Remediation**

We suggest changing the code so only token holders can burn their own tokens and not anyone else. Not even a contract creator.

**Status**

Fixed

# Medium Severity Issues

### C.2 Uncheck transfer

**Description**

The return value of an external transfer/transferFrom call is not checked since the external tokens do not revert in case of failure and return false. We've found the following return values are not checked.
L621: quoteToken.transferFrom(msg.sender, address(this), _amountQuote);

**Remediation**

Please consider adding the require() check for those external calls or using SafeERC20, or ensure that the transfer/transferFrom return value is checked.

**Status**

Fixed

## Low Severity Issues

### C.3 Lack of event emissions

**Description**

The missing event makes it difficult to track off-chain liquidity fee changes. An event should be emitted for significant transactions calling the following functions:

- setPaused()
- setOperator()
- setRouter()
- setPriceProtection()
- setTaxAddress()
- setTaxRate()
- setTaxExcluded()
- setTaxTransferTypeExcluded()()

**Recommendation**

We recommend emitting an event to log the update of the above variables for those above-mentioned functions.

**Status**

**Acknowledged**

### C.4 Lack of Zero address validation

**Description**

To favor explicitness, consider adding a check for all functions that are taking address parameters in the entire codebase. Although most of the functions throughout the codebase properly validate function inputs, there are some instances of functions that do not. One example is:

- setTaxAddress()
- rescue()

**Recommendation**

Consider implementing require statements where appropriate to validate all user-controlled input, including governance functions, to avoid the potential for erroneous values to result in unexpected behaviors or wasted gas.

**Status**

**Acknowledged**

## Informational Issues

### C.5 Public function that could be declared external

**Description**

The following public functions that are never called by the contract should be declared external to save gas:

- setPaused()
- setOperator()
- setRouter()
- mint()
- setPriceProtection()
- setTaxAddress()
- setTaxRate()
- setTaxExcluded()
- setTaxTransferTypeExcluded()
- selfApprove()
- transferNoTax()
- addLiquidity()
- removeLiquidity()
- rescue()
- setWhitelistLock()
- setSwapWhitelist()
- setBlocklist()

**Recommendation**

Use the external attribute for functions never called from the contract.

**Status**

**Acknowledged**

## C.6 Floating Pragma

### Description

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Using floating pragma does not ensure that the contracts will be deployed with the same version. It is possible that the most recent compiler version get selected while deploying contract which has higher chances of having bugs in it.

### Remediation

Lock the pragma version for the compiler version that is chosen.

### Status

**Acknowledged**

## D. Contract - Referral.sol

## High Severity Issues

No Issues Found

## Medium Severity Issues

### D.1 transfer() is being utilized

**Description**

Low-level transfer() function has been found to be used in the contract at line 158:

_to.transfer(_amount);

Due to the fact that .transfer() and .send() forward exactly 2,300 gas to the recipient. This hardcoded gas stipend aimed to prevent reentrancy vulnerabilities, but this only makes sense under the assumption that gas costs are constant. Recently EIP 1884 was included in the Istanbul hard fork. One of the changes included in EIP 1884 is an increase to the gas cost of the SLOAD operation, causing a contract's fallback function to cost more than 2300 gas.

```
// bad
contract Vulnerable {
    function withdraw(uint256 amount) external {
        // This forwards 2300 gas, which may not be enough if the recipient
        // is a contract and gas costs change.
        msg.sender.transfer(amount);
    }
}

// good
contract Fixed {
    function withdraw(uint256 amount) external {
        // This forwards all available gas. Be sure to check the return value!
        (bool success, ) = msg.sender.call.value(amount)("");
        require(success, "Transfer failed.");
    }
}
```

**Remediation**

The auditee needs to ensure that the _to is not a contract .On the other hand, it's recommended to stop using .transfer() and .send() and instead use .call().

**Status**

**Fixed**

## Low Severity Issues

### D.2 Lack of event emissions

**Description**

The missing event makes it difficult to track off-chain liquidity fee changes. An event should be emitted for significant transactions calling the following functions:
- setOperator()
- setConfig()
- setRewardRates()
- setThisPeriod()

**Recommendation**

We recommend emitting an event to log the update of the above variables for those above-mentioned functions.

**Status**

**Acknowledged**

### D.3 Use of extcodesize can be exploited

**Description**

Referral contract uses isContract function at [#L100] !_from.isContract(),For checking the address is contract or not the isContract() function uses extcodesize opcode which may be circumvented by a contract address during construction when it does not have source code available.

**Recommendation**

We recommend reviewing the logic. In the case if the address is known for which check is being made, use a conditional statement to check if the the address is the one for which the check is being made.

**Status**

**Acknowledged**

## Informational Issues

### D.4 Public function that could be declared external

**Description**

The following public functions that are never called by the contract should be declared external to save gas:

- setPaused()
- setOperator()
- setTokenAddress()
- rescue()

**Recommendation**

Use the external attribute for functions never called from the contract.

**Status**

**Acknowledged**

### D.5 Floating Pragma

**Description**

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Using floating pragma does not ensure that the contracts will be deployed with the same version. It is possible that the most recent compiler version get selected while deploying contract which has higher chances of having bugs in it.

**Remediation**

Lock the pragma version for the compiler version that is chosen.

**Status**

**Acknowledged**

## D.6 Dead Code

### Description

The condition _amount >= minReturnReferralAmount is getting checked twice. on the [#L98] as the if statement's second condition _amount < minReturnReferralAmount performs the first check. And [#L113] _amount >= minReturnReferralAmount is the second check for the same.

### Remediation

Consider removing the extra check on line 113.

### Status

**Acknowledged**

# Functional Testing

**Contract - Farm.sol**

**Testing setter functions**

- ✓ setPaused() should be called only by the owner
- ✓ setOperator() should be called only by the owner
- ✓ setOperator() returns true if all passed
- ✓ setTokenAddress() should be called only by the onwer
- ✓ setTokenAddress() should return true if all passed
- ✓ addPool() should be called only by the owner
- ✓ addPool() for pid = 0 with _isLinearRelease && _shareWithPreDeposit == True
- ✓ addPool() for pid = 1 with _isLinearRelease == True && _shareWithPreDeposit == False
- ✓ addPool() for pid = 2 with _isLinearRelease == False && _shareWithPreDeposit == True
- ✓ addPool() for pid = 3 with _isLinearRelease == False && _shareWithPreDeposit == False
- ✓ addPool() for pid = 4 with _isLinearRelease == False && _shareWithPreDeposit == False
- ✓ addPool() for pid = 5 with _allocPoint == 0
- ✓ setPool() should be called only by the owner
- ✓ pool() should return correct values of pid = 4 before setPool() is called
- ✓ setPool() should true when update made on the pid = 4
- ✓ pool() should return correct values of pid = 4 after setPool() is called
- ✓ setPreDepositPid() should be called only by the owner
- ✓ setPreDepositPid() should revert if _pid >= _pools.length
- ✓ setPreDepositPid() set pid = 4 as a PreDepositPid
- ✓ addReward() should be called only by the owner
- ✓ Cross check mainRewardPending() and lpRewardPending() before addReward()
- ✓ deposit() should be reverted when paused = True
- ✓ user1 calls deposit() should get reverted when _pid == preDepositPid
- ✓ user1 calls deposit() should get reverted when _pid > _pools.length
- ✓ user1 calls deposit() should get reverted when amount > approved amount
- ✓ user1 calls deposit() successfully with some amount =< approved amount
- ✓ userInfo() should return correct values of user1
- ✓ pool[0]() should return correct values of pid = 1
- ✓ getUserDepositAmount() of the user1 should return a correct value for a single deposit
- ✓ getDepositAmount() should return a correct value for a single deposit
- ✓ other main reward and lp reward of the user1 should be Zero before dispersed addeReward()
- ✓ addReward() should return True when called by operator
- ✓ addReward() should return True when called by the current owner
- ✓ Cross check mainRewardPending() and lpRewardPending() after addReward()

- ✓ disperseMainReward() should be called only by the operator
- ✓ Cross check mainRewardDispersed() before disperseMainReward() is called
- ✓ Cross check mainRewardPending() and mainRewardDispersed() before disperseMainReward() is called (64ms)
- ✓ disperseMainReward() is called successfully by the operator
- ✓ Cross check mainRewardPending() and mainRewardDispersed() after disperseMainReward() is called
- ✓ disperseLpReward() should be called only by the owner
- ✓ Cross check lpRewardDispersed() before disperseLpReward() is called
- ✓ Cross check lpRewardPending() and lpRewardDispersed() before disperseLpReward() is called (58ms)
- ✓ disperseLpReward() is called successfully by the operator
- ✓ Cross check lpRewardPending() and lpRewardDispersed() after disperseLpReward() is called
- ✓ userInfo() should be updated after disperseLpReward() and disperseMainReward() are called
- ✓ preDeposit() should be called only by the owner
- ✓ preDeposit() is called by the operator to user2 (62ms)
- ✓ Cross check userInfo() of user2
- ✓ multiPreDeposit() should be called only by the operator
- ✓ The operator calls multiPreDeposit() (71ms)
- ✓ Cross check userInfo() of user3, user4, user 5 in the array
- ✓ Check canWithdrawAmount() of all users (60ms)
- ✓ Users can't deposit when paused == true
- ✓ User1 calls deposit() to the pool == 2 after paused == false (43ms)
- ✓ Amount returned by canWithdrawAmount() of user1 should be changed when time passed, while isLinearRelease == false (39ms)
- ✓ Amount returned by canWithdrawAmount() of user3 should be changed when time all passed, while isLinearRelease == true (39ms)
- ✓ takeReward() fails when paused == true
- ✓ [Failed to the business logic] user 1 calls takeReward() before disperseMainReward() and disperseLpReward() (50ms)
- ✓ [Failed to the business logic] user 1 calls takeReward() after disperseMainReward() and disperseLpReward() (115ms)
- ✓ withdraw() fails when paused == true
- ✓ withdraw() should revert when _pid >= _pools.length,
- ✓ withdraw() should revert when _amount > _canWithdrawAmount,
- ✓ user1 calls withdraw() should be successful, (74ms)

- ✓ User6 adds some amount to the pool1
- ✓ user1 calls withdraw() should be successful, (153ms)

**Testing setter functions**

- ✓ owner() should be correct
- ✓ operators() should return true for the current owner
- ✓ operators() should return true for the new operator
- ✓ depositToken() should return true
- ✓ mainRewardToken() should return true
- ✓ lpRewardToken() should return true
- ✓ pool() should revert if _pid > _pools.length
- ✓ pool() should return correct values of pid = 0
- ✓ pool() should return correct values of pid = 1
- ✓ pool() should return correct values of pid = 2
- ✓ pool() should return correct values of pid = 3
- ✓ poolLength() should return true
- ✓ updateTotalAllocPoint() should work and update via totalAllocPoint(), it should return a correct value

**Contract - Lottery.sol**

- ✓ Should be able to initialize variables with setConfig()
- ✓ Should be able to set reward rates
- ✓ setRewardRates Reverts if addition of rates is greater than 10000
- ✓ onTransfer executes fully only for transfer type 1
- ✓ onTransfer() adds new active orders succesfully
- ✓ onTransfer() removes first orders if length of active orders is greater than length of reward rates
- ✓ draw() sorts the active orders by quote amount and order timestamp
- ✓ draw() dipserses the rewards for active orders
- ✓ takeReward() allows to take reward for whole duration
- ✓ takeReward() allows to take reward for completed duration before release hours
- ✓ setPaused() should be called only by the owner (44ms)
- ✓ setPaused() can be set to True or False
- ✓ setOperator() should be called only by the owner
- ✓ setOperator() sets operator user
- ✓ setOperator() sets N3DR as an operator
- ✓ setConfig() should be called only by the owner
- ✓ setConfig() is called successfully by the owner
- ✓ setRewardRates() should be called only by the owner
- ✓ setRewardRates() should revert if _rates > 10000
- ✓ setRewardRates() set by the owner successfully
- ✓ pendingReward() should return balance of Lottery
- ✓ pendingReward() after lockedReward() is set
- ✓ Should return mainToken()
- ✓ Should return rewardRates() (46ms)
- ✓ Should return rewardRatesLength()

**Contract - N3DR.sol**

- ✓ setRouter() should be called only by the owner (38ms)
- ✓ setRouter() calls by admin (44ms)
- ✓ setPaused() should be called only by the owner
- ✓ setPaused() can be set to True or False
- ✓ setOperator() should be called only by the owner
- ✓ setOperator() sets operator user
- ✓ setOperator() sets farm, lottery and referral as the operators (57ms)
- ✓ selfApprove() should be called by the owner
- ✓ selfApprove() set for n3dr and Router
- ✓ selfApprove() set for quoteToken and Router
- ✓ mint() for admin address should be called only by the owner
- ✓ setPriceProtection() should be called only by the owner
- ✓ setTaxAddress() should be called only by the owner
- ✓ setTaxRate() should be called only by the owner
- ✓ setTaxExcluded() should be called only by the owner (74ms)
- ✓ setTaxTransferTypeExcluded() should be called only by the owner (42ms)
- ✓ Owner mint some token for user2
- ✓ Owner should not burn other user token
- ✓ addLiquidity() for 2 tokens
- ✓ SelfApprove gives approval for spending n3dr tokens to spender
- ✓ transferNoTax transfers tokens without deducting tax
- ✓ transferNoTax updates todayTimeIndex and todayOpenPrice once liquidity added to pool
- ✓ _transfer() deducts taxes
- ✓ _transfer() doesn't deducts taxes when address is excluded
- ✓ _transfer() doesn't deducts taxes when transfer type is excluded
- ✓ _transfer() udpates lastBuyTime if transfer type is 1
- ✓ _transfer() transfers 99% amount of balance for transfer type 2 and when amount to transfer is greater or equal to balance of sender
- ✓ _transfer() deducts additional sell tax if daily price drop rate > 20%
- ✓ _transfer() reverts if lastBuyTime of from address is not greater than 0
- ✓ _transfer() reverts if token holding time is too short
- ✓ addLiquidity() adds liquidity to pool
- ✓ removeLiquidity() removes liquidity from pool
- ✓ rescue() transfers native tokens to _to address
- ✓ rescue() transfers bep20 tokens to _to address
- ✓ getPrice() gives quotetoken price for 1 n3dr token
- ✓ getDailyPriceChange() gives change rate in price

**Contract - Referral.sol**

- ✓ setConfig() should be called only by the owner
- ✓ setOperator() should be called only by the owner
- ✓ setRewardRates() should be called only by the owner
- ✓ setThisPeriod() should be called only by the owner
- ✓ onTransfer() should be called only by the operator
- ✓ addReward() should be called only by the operator
- ✓ rewardRatesLength() should return the correct length of the rewardRates
- ✓ referralUserLength() should return the correct length of the usersOfReferrer
- ✓ userLength() should return the correct length of the _rewardUsers()
- ✓ users() should return its information within an interval
- ✓ userRewards() should return its information within an interval
- ✓ userRewardsOfPeriod() should return its information within an interval
- ✓ isActiveUser() should check if mainToken.getLpValue(_user) >= minActiveReferralValue
- ✓ countActiveUsers() should check if isActiveUser(users_[index])
- ✓ rescue() should be called only by the owner
- ✓ setConfig() should be able to initialize variables
- ✓ setConfig() Reverts if called by unauthorized address
- ✓ setRewardRates() Reverts if called by unauthorized address
- ✓ setRewardRates() Reverts if addition of reward rates is greater than referral tax amount in n3dr contract
- ✓ setRewardRates() Reverts if 0th value of reward rates is not zero
- ✓ setRewardRates () Should be able to set the reward rates
- ✓ setOperator() should be able to set operator()
- ✓ setOperator() Reverts if caller is not owner
- ✓ setThisPeriod() Should be able to set thisPeriod
- ✓ setThisPeriod() Should revert if caller is not owner
- ✓ onTransfer() adds referrer of user
- ✓ onTransfer() adds users of referrer
- ✓ addReward() distributes rewards to referrals from _rewardAmount and sends remaining to lottery contract address

# 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 the NeorderDao platform. We performed our audit according to the procedure described above.

The audit showed several high, medium, low, and informational severity issues. In the end, the majority of the issues were fixed and acknowledged by the Auditee.

# Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Neorder Platform. 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 Neorder Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

# About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

**500+**
Audits Completed

**$15B**
Secured

**500K**
Lines of Code Audited

## Follow Our Journey

# Audit Report
# April, 2022

For

Neorder

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com